저작자표시-비영리-변경금지 2.0 대한민국

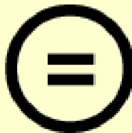
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

M.S. THESIS

LSM-tree based Database System Optimization using Application-Driven Flash Management

응용 프로그램 기반의 플래시 저장 장치 관리 기법을 통한 LSM-tree 기반 데이터베이스 성능 최적화

BY

임 희 락

AUGUST 2019

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

M.S. THESIS

LSM-tree based Database System Optimization using Application-Driven Flash Management

응용 프로그램 기반의 플래시 저장 장치 관리 기법을 통한 LSM-tree 기반 데이터베이스 성능 최적화

BY

임 희 락

AUGUST 2019

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

LSM-tree based Database System Optimization using Application-Driven Flash Management

응용 프로그램 기반의 플래시 저장 장치 관리 기법을 통한 LSM-tree 기반 데이터베이스 성능 최적화

지도교수 염 헌 영

이 논문을 공학석사 학위논문으로 제출함

2019 년 04 월

서울대학교 대학원

컴퓨터 공학부

임 희 락

임 희 락의 공학석사 학위논문을 인준함

2019 년 06 월

위 원 장	신영길	(인)
부위원장	염헌영	(인)
위 원	엄현상	(인)

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 key-value stores. Key-value stores are the most typical applications that should provide a highly reliable service with high-performance. To increase the I/O performance of key-value stores, many data centers have actively adopted next-generation storage devices such as Non-Volatile Memory Express (NVMe) based Solid State Devices (SSDs). NVMe SSDs and its protocol are characterized to provide a high degree of parallelism. However, they may not guarantee predictable performance while providing high performance and parallelism. For example, heavily mixed read and write requests can result in performance degradation of throughput and response time due to the interference between the requests and internal operations (e.g., Garbage Collection (GC)).

To minimize the interference and provide higher performance, this paper presents IsoKV, an isolation scheme for key-value stores by exploiting internal parallelism in SSDs. IsoKV manages the level of parallelism of SSD directly by running application-driven flash management scheme. By storing data with different characteristics in each dedicated internal parallel units of SSD, IsoKV reduces interference between I/O requests. We implement IsoKV on RocksDB and evaluate it using Open-Channel SSD. Our extensive experiments have shown that IsoKV improves overall throughput and response time on average 1.20× and 43% compared with the existing scheme, respectively.

Keywords: Storage, NAND-flash, Open-Channel-SSD, FTL, NVMe, LSM-tree
Student Number: 2017-21118

Contents

Abstract **i**

Chapter 1 Introduction **1**

Chapter 2 Background **8**

2.1 Log-Structured Merge tree based Database 8

2.2 Open-Channel SSDs 9

2.3 Preliminary Experimental Evaluation using oc_bench 10

Chapter 3 Design and Implementation **14**

3.1 Overview of IsoKV 14

3.2 GC-free flash storage management synchronized with LSM-tree logic 15

3.3 I/O type Isolation through Application-Driven Flash Management 17

3.4 Dynamic Arrangement of NAND-Flash Parallelism 19

3.5 Implementation 21

Chapter 4 Evaluation **23**

4.1 Experimental Setup 23

4.2 Performance Evaluation 25

Chapter 5 Related Work **31**

Chapter 6 Conclusion **34**

Bibliography **35**

초록 **40**

List of Figures

Figure 1.1 Hardware Queue Contention in NVMe SSD. 2

Figure 1.2 Read and write throughput comparison based on various read percentage in the workload. 4

Figure 2.1 Comparing architectures when read and write requests are physically isolated and processed in Open-Channel SSD using the *oc_benc tool*. 10

Figure 2.2 oc_bench - Read, Write, Overall throughtput performance comparison 13

Figure 3.1 life cycle of a virtual block (vblk). This figure shows the state of vblk and the transition between the states. Depending on the behavior of the LSM-tree logic, the state of the application layer data (eg, WAL, SST) is synchronized with the physical blocks of NAND flash. Therefore, it is guaranteed that the physical blocks that have stored the outdated data which is deleted in the application are free. 16

Figure 3.2 IsoKV storage backend scheme using Open-Channel SSD and liblightnvm 20

Figure 4.1 Striping-Arrangement VS Isolation-Arrangement. 24

Figure 4.2 Average Read Latency of key-value pair from Level-0 to Level-3 SST files 27

Figure 4.3 $P99^{th}$ Tail Latency of key-value pair from Level-0 to Level-3 28

Figure 4.4 Interval throughput comparison under changing workload - Striping VS Isolation VS Dynamic 29

Figure 4.5 YCSB Macro-benchmarks result 30

List of Tables

Table 4.1 Workload descriptions and parameters 29

Chapter 1

Introduction

The role of high-performance storage devices is becoming indispensable to I/O intensive applications (e.g., key-value stores) since the performance of storage devices directly affects their quality of service (QoS). Accordingly, many data centers have actively adopted next-generation storage devices such as NVMe SSDs to improve QoS and I/O performance. NVMe SSDs provides higher read and write throughput compared with spinning drives or SATA-based SSDs. However, modern applications or systems (e.g., key-value database [2, 3, 4, 5]) may often not fully utilize the capabilities of the NVMe SSD devices. The reason is that they are not designed with a thorough consideration of the features of the NVMe SSD storage devices.

For example, serious performance degradation may be caused when write and read operations are performed simultaneously. Performance degradation could be up to 450% under the workloads with mixed reads and writes as mentioned in previous studies [15, 16]. It is because the processing time for write operations is longer than that of read operations due to internal FTL

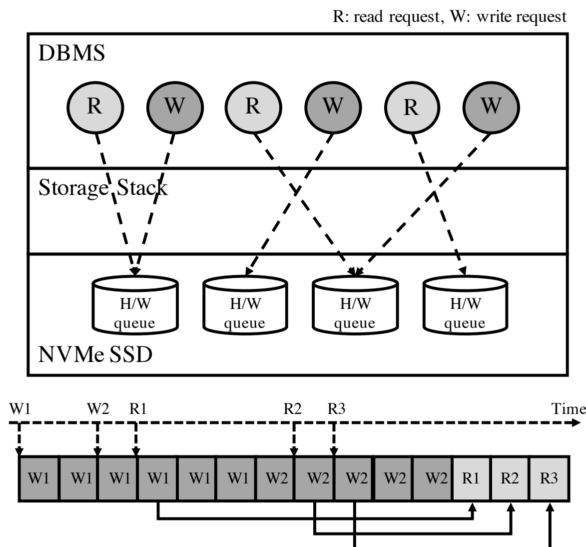


Figure 1.1: Hardware Queue Contention in NVMe SSD.

operations such as Garbage Collection (GC) and flushing buffer cache.

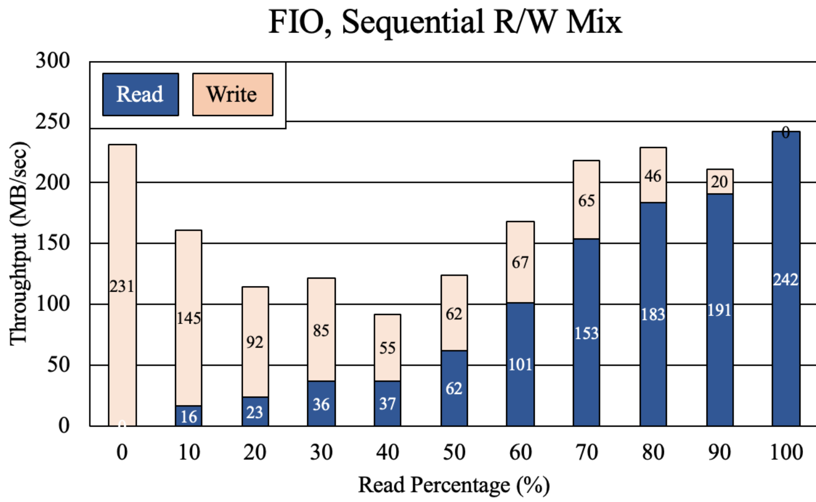
As shown in Figure 1.1, if both read and write requests arrive at the same H/W queue in NVMe SSD, contention occurs so that read requests are delayed due to the write requests that require long response time. Furthermore, unlike ordinary relational database systems, in the LSM-tree based key-value store, additional I/O requests are generated due to the compaction that creates higher-level Sorted String Table (SST) files by merging lower-level SST files. For example, even if clients only send insert requests, underlying flash-based storage devices must handle an amplified number of write requests 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 NVMe based SSDs and LSM-tree based Key-

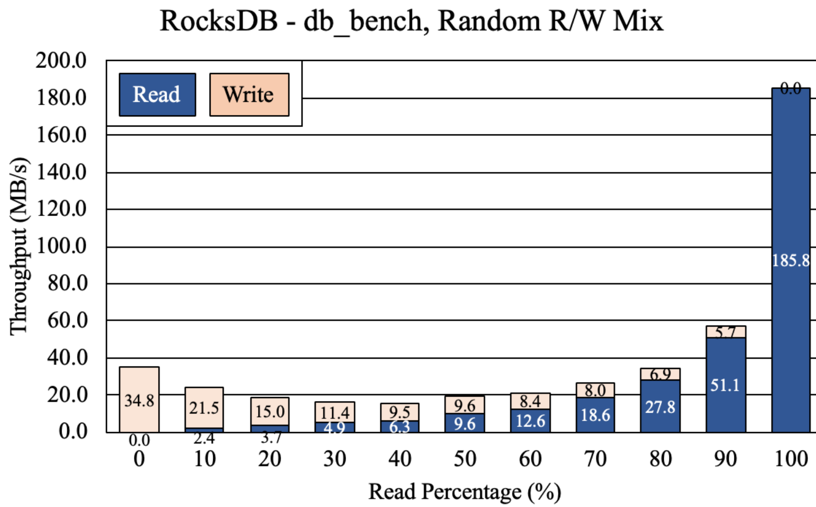
Value store on commercial NVMe SSDs [10], we used a Flexible I/O (FIO) benchmark tool [1] and RocksDB db_bench tool [9] respectively. Figure 1.2a represents the result shows serious performance degradation depending on the degree of mixed reads and writes. In particular, under the workload that read:write ratio is 50:50, the read performance decreased by 74.4% compared with the read performance under the read-only workload. Also, the write performance dropped to about a quarter of the write-only performance. This graph shows a dramatic drop in performance depending on the degree of mixed reads and writes. Figure 1.2b illustrates that performance degradation due to I/O interference is much more serious in RocksDB. Even if write requests occupy only 10 percent and the read request accounts for 90 percent of the total workload, the read performance drops to a quarter compared to the read-only performance. The reason for this result comes from write amplification due to the nature of the LSM-tree based key-value store; underlying storage device must accommodate larger number of I/O requests than the client issues.

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 an 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. We have found that this scheme does not guarantee predictable and high performance.

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



(a) FIO benchmark



(b) RocksDB db_bench tool

Figure 1.2: Read and write throughput comparison based on various read percentage in the workload.

are listed as follows:

Preliminary experimental evaluation using oc_bench. We design and implement FIO-like benchmark tool called *oc_bench* [8] tailored Open-Channel SSD. *oc_bench* is a tool for evaluating device performance, changing many parameters, such as the number of threads and file sizes like existing benchmark tools (e.g., FIO, Iometer). In addition, *oc_bench* can determine the physical address of data to be stored by utilizing the features of an Open-Channel SSD. We evaluate 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 has their own region without the interference of other I/O threads by dedicated Parallel Units (LUNs) per I/O thread. The evaluation result demonstrates the overall performance is improved by up to 220% compared with the baseline that each thread uses entire LUNs greedily.

GC-free flash storage management synchronized with LSM-tree logic.

We synchronize the data life cycle in LSM-tree logic with the life cycle of data in the SSD. That is, each SST file and WAL log file are managed by using vblk in the storage which is directly mapped with physical blocks. If a file is created, the corresponding type of vblk allocated for it. According to the LSM-tree logic, when a file is outdated due to compaction (SST file) or completion of flushing memtable (WAL file), the vblk is released and become free. As a result, this eliminates the GC that the SSD unnecessarily copies valid data into a new block.

I/O type isolation through application-driven flash management. In order to apply the read/write isolation scheme in *oc_bench* to IsoKV, we modify the RocksDB storage backend so that the write requests of different

I/O types (e.g., log file, Level 0 SST file, etc.) does not physically overlap within the NAND Flash each other. Unlike the micro-benchmark tool (e.g., FIO or *oc_bench*), database applications must read data at the specific location where the data is written. Thus, IsoKV cannot completely isolate read requests from write requests in the SSD physically. However, the characteristics of LSM-tree base database storing data as append-only and the write-only files (Write Ahead Log) can mitigate the contention of reads and writes in runtime.

Dynamic arrangement of NAND-flash parallelism. Under workloads that read and write requests are frequently conflicted, Isolation-Arrangement scheme of IsoKV shows improved performance, but in other workloads such as read-only or write-only, existing scheme that makes the most of parallelism performs better than our IsoKV. To ensure that IsoKV is flexible and work best for all workloads, we add dynamic LUNs arrangement scheme to IsoKV. Under the write-intensive workload that relatively requests are not mixed frequently, we make each I/O type of IsoKV utilize entire LUNs to support high parallelism. To determine the arrangement of LUNs by analyzing the nature of the workload at runtime, we designed a simple count-based workload profiler on IsoKV's storage backend.

Evaluating IsoKV with realistic workloads. We carefully evaluated IsoKV under various read:write ratio using microbenchmark tool. The result represents that IsoKV reduces average read response time by 43% compared to the baseline that works in a greedy manner. We also verified the flexibility of IsoKV by running a workload in which nature is changed during runtime. The Evaluation result shows that IsoKV responds flexibly to the changing workload and performed better than static LUNs arrangement scheme. Furthermore, we run YCSB benchmark to evaluate IsoKV under realistic workloads. Under four of YCSB's representative workloads showed improved performance, up to 1.56x

faster overall throughput and 60% reduced read response time. The open-source IsoKV is available at *https://github.com/RockyLim92/IsoKV*.

Chapter 2

Background

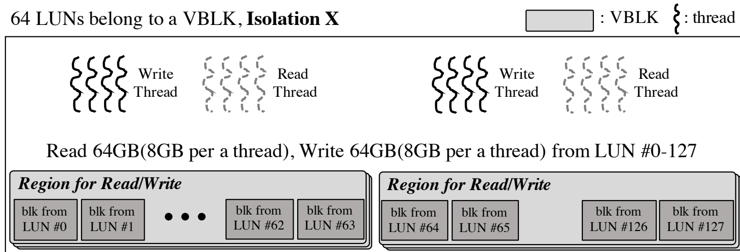
2.1 Log-Structured Merge tree based Database

LSM-tree data structure is used in many modern key-value stores and storage systems to provide fast I/O services [4, 3, 5, 2]. It stores data in an append-only manner and thus provides a fast write performance. In more detail, the incoming key-value pairs from clients are written in sorted order in the in-memory write buffer called memtable. If data is written as much as the allocated size of memtable, background flush threads flush data in 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, Sum of level 0 SST files size is exceeded a predefined threshold and this spoils the shape of the LSM-tree data structure. Then, a background compaction mechanism is triggered to constrain the LSM tree shape. It reads multiple L0 SST files to merge them into a next level SST file. This compaction mechanism works in the same way at level 0 as well as at other levels.

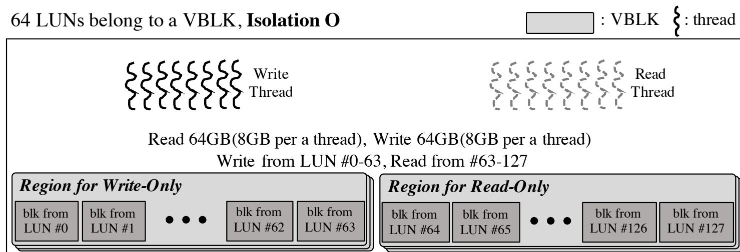
While compaction, there may be multiple duplicated key-value pairs among the input SST files of compaction. Only one valid key remains in the compaction output SST file, and the remaining invalid key-value pairs are removed. Through this compaction mechanism, LSM-tree-based database can store key-value pairs in append-only manner without in-place update. In RocksDB, our target application, each different I/O type accesses to the storage device by different threads. For example, foreground I/O threads append log data to the WAL log file. On the other hand, the *L0 SST files* and the *L* (* > 0)SST files* are flushed by background flush threads and background compaction threads respectively.

2.2 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 [41, 42, 6]. They expose the geometry inside the SSD and share control responsibilities with the host 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 or manage the NAND-flash based storage device at host side. In order to allow the host application to interact with Open-Channel SSDs from user space, Open-Channel SSD provides a user space I/O library, a *liblightnvm* [7]. The core of the library provides an interface for performing vectorized I/O using physical addressing. *virtual block (vblk)* interface of liblightnvm provides libc-like *write, read, pread* interface, encapsulating command-construction including mapping a general physical addressing format to device specific. Thereby allowing the user to focus on performing read/write on the vector space. A vblk consists of a set of physical



(a) read/write reuqests are mixed



(b) read/write requests are isolated each other

Figure 2.1: Comparing architectures when read and write requests are physically isolated and processed in Open-Channel SSD using the *oc_benc tool*.

blocks in the SSD and can be created to span all parallel units (LUNs) of SSD or a subset thereof. For example, in Open-Channel SSD equipped with 16 channels and 8 dies(chips) per channel, an I/O thread can access up to 128 independent blocks at once using vblk which consists of 128 blocks from each of 128 LUNs.

2.3 Preliminary Experimental Evaluation using oc_bench

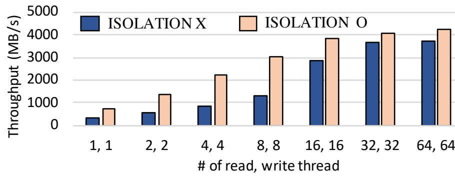
We found the performance degradation due to the interference between read and write requests through FIO experiments using commercial NVMe SSDs.

In this chapter, we demonstrate how performance can be improved when the interference is eliminated by using Open-Channel SSDs.

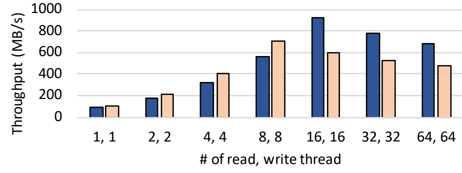
We developed oc_bench [8], an FIO-like benchmark tool tailored for Open-Channel SSDs. oc_bench spawns a number of thread doing a particular type of I/O action as specified by the user. Especially, it uses a fvirtual block (vblk) interface to access the Open-Channel SSD and it is able to set which thread will perform I/O to which physical address (i.e., channel, die, block, etc.). 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. Figure 2.1 shows an example that how the oc_bench works when a vblk is constructed by taking a block from each of 64 LUNs. Under the configuration that read and write requests share the same region (2.1a), a NAND flash chip belonging to a specific LUN serves both read and write requests, thus causing I/O interference. However, in the isolated design as shown in Figure 2.1b, the interference does not occur because only reads or writes occur in each LUN.

In addition, we run oc_bench which issues interleaved read and write requests to the Open-Channel SSD The dataset size for the experiment is 40 GB, and oc_bench generates read and write requests at the ratio of 50%. We evaluated the raw device performance under the workload and varying 42 configurations depending on the presence of isolation, the number of threads, and the size of vblk. Figure 2.2a/2.2c, Figure 2.2d/2.2f, and Figure 2.2g/2.2i represent the results of evaluation under the varying vblk size, 8 LUNs, 16 LUNs and 32 LUNs, respectively. As shown in figures, the isolation scheme improve the read performance on all vblk sizes, except for the configuration where the number of read and write threads are 64 each. This is due to overall performance degradation due to excessive I/O contention, rather than the cause associated with the isolation scheme. As shown in Figure 2.2b, Figure 2.2e and Figure 2.2h,

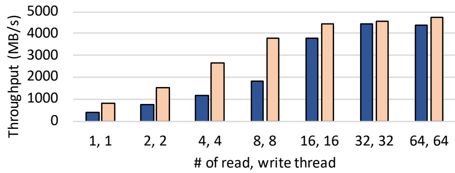
the write performance was also improved by the reduced interference. However, the write performance is degraded when multiple write threads try to access the same virtual block at the same time. For example, when 32 LUNs consist of a single vblk and more than two write threads exist in the isolation scheme, a contention occurs in vblk and performance degrades. 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 they access to vblks. Therefore, if the smaller number of LUNs configuring a vblk is used, each thread may access to the isolated LUN so that the result shows better scalability and performance. Figure 2.2c, Figure 2.2f and Figure 2.2i illustrate that overall performance was improved in most configurations. In particular, there was a performance improvement of up to 220% in configurations where there is no excessive contention of write threads in vblk. This performance gain is attributed to the significant performance improvement in reads due to the reduced I/O interference.



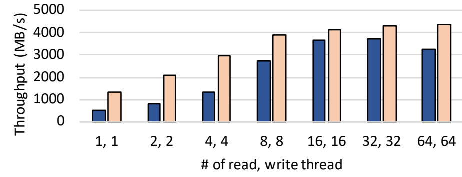
(a) Read throughput, 8 LUNs construct a vblk



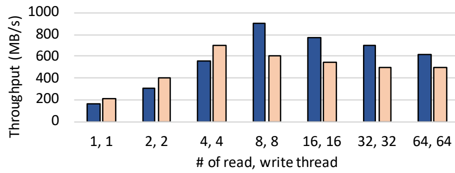
(b) Write throughput, 8 LUNs construct a vblk



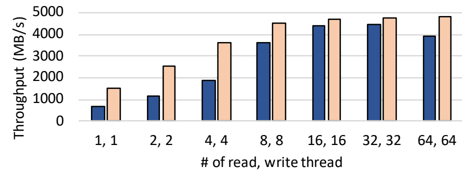
(c) Overall throughput, 8 LUNs construct a vblk



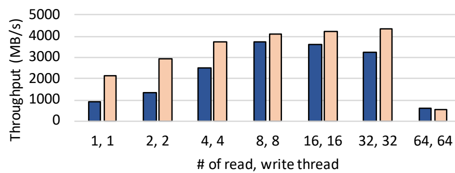
(d) Read throughput, 16 LUNs construct a vblk



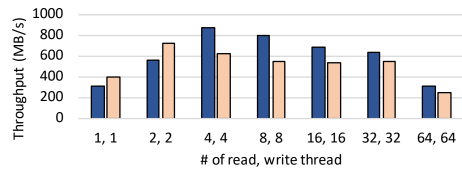
(e) Write throughput, 16 LUNs construct a vblk



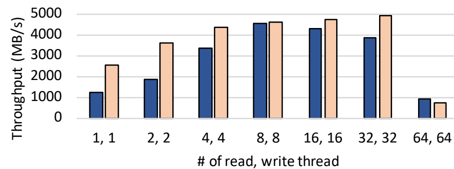
(f) Overall throughput, 16 LUNs construct a vblk



(g) Read throughput, 32 LUNs construct a vblk



(h) Write throughput, 32 LUNs construct a vblk



(i) Overall throughput, 32 LUNs construct a vblk

Figure 2.2: oc_bench - Read, Write, Overall throughtput performance comparison

Chapter 3

Design and Implementation

3.1 Overview of IsoKV

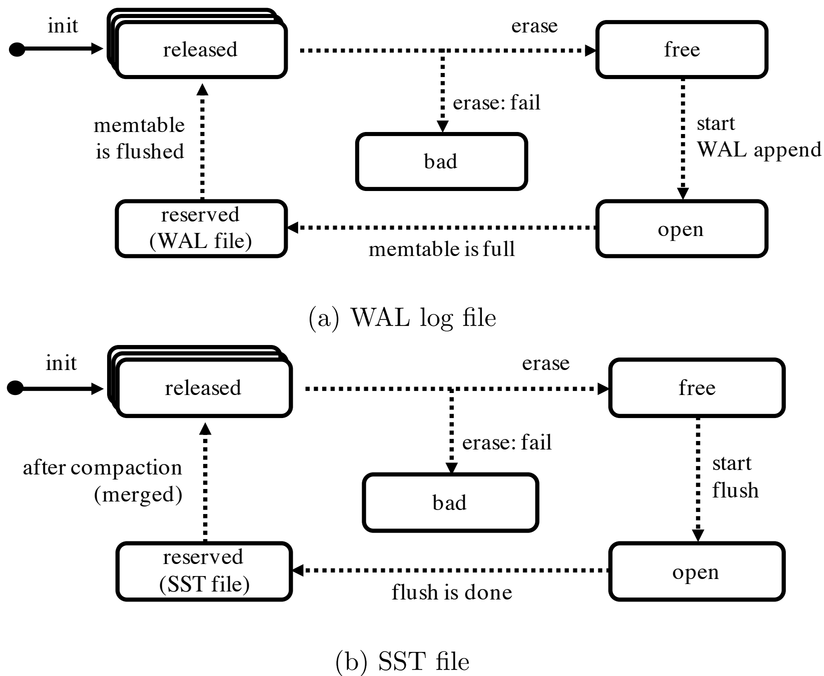
In this section, we describe IsoKV, three key design approaches, and implementation details to eliminate interference between reads and writes for achieving higher performance. IsoKV is an LSM-tree based KV store which has application-driven flash management scheme using Open-Channel SSDs as it's underlying storage device. It achieves both high throughput performance and reduced latency. At first, we propose a GC-free flash storage management scheme, in which flash blocks which store data with the same life cycle are managed together and have a life cycle that is synchronized with the application's data life cycle. Second, we design a new data placement policy which isolates and stores different type of application data into separate physical flash storage units (i.e., Parallel Units(LUNs)). Finally, we propose a dynamic arrangement scheme for data placement to adaptively cope with various workloads.

3.2 GC-free flash storage management synchronized with LSM-tree logic

Many previous studies have shown that the internal operations of SSD degrade the predictability of foreground I/O performance. In particular, GC blocks incoming I/O and degrades overall storage performance [35, 36, 37]. In this section, we introduce the GC-free flash storage management scheme of IsoKV, which considers LSM-tree logic and I/O components. IsoKV synchronizes the life cycle of NAND flash blocks of SSD with the life cycle of data according to the LSM-tee logic In IsoKV, all files created during the operation of the KV store are stored using liblightnvm, the user space I/O library for OCSSD. The vblk interface of liblightnvm groups the physical blocks of the Open-Channel SSD to form a vblk using vblk interface, I/O requests bypasse the existing file system or block layer to read and write data. Figure 3.1a and Figure 3.1b show an example of a life cycle synchronized with each I/O component and storage in IsoKV. When initilalizing IsoKV, different types of vblks are created according to a predetermined degree of parallelism.

The life cycle of the WAL file starts with erasing the released vblk. vblk which completes the erase start append data for recovery. Once the member table is full, and then it is flushed to the SST file, the WAL file is no longer needed. Therefore, the corresponding vblk is released.

On the other hand, the life cycle of an SST file starts when memtable begins the flush operation. IsoKV determines which vblk should be allocated according to whether the new SST table is Level 0 SST (flush) or Level 1 SST (compaction output) or more. This is covered in more detail in the next section. When IsoKV flushes memtable and creates an immutable SST file, the corresponding vblk becomes reserved. If an SST file is selected as a compaction target due to the



(a) WAL log file

(b) SST file

Figure 3.1: life cycle of a virtual block (vblk). This figure shows the state of vblk and the transition between the states. Depending on the behavior of the LSM-tree logic, the state of the application layer data (eg, WAL, SST) is synchronized with the physical blocks of NAND flash. Therefore, it is guaranteed that the physical blocks that have stored the outdated data which is deleted in the application are free.

insertion of subsequent data, valid key-value data of the SST file is copied to memory and merged into the source of the new SST file. Then the vblk, which stores the outdated SST file, is released.

3.3 I/O type Isolation through Application-Driven Flash Management

Through preliminary experiment results, we found that utilizing an SSD device's entire parallelism does not always lead to the best performance. Hence, we implemented IsoKV based on RockDB, to apply this characteristic 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 is written.

However, as shown in Figure 3.2a and Figure 3.2b, LSM-tree based database including RocksDB has different I/O scheme compared with ordinary Database Management Systems (DBMS) or storage systems. In RocksDB, three types of files are written to the persistent storage device by key-value insertions and periodical compaction algorithm. First of all, 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 algorithm. For the simplest, since WAL is write-only, IsoKV can allocate a completely isolated LUNs for WAL files so that other threads are free from interference due to the WAL. 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 a read-only manner.

Figure 3.2a shows a baseline, a Striping-Arrangement scheme of IsoKV, to compare with our isolated design. In this architecture, all different types of IsoKV shares 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 3.2b represents the IsoKV architecture with our optimization applied, an 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 types can be isolated from 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.

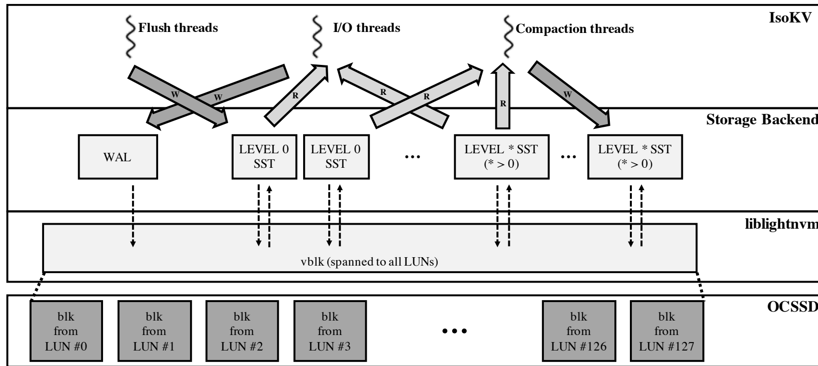
A vblk is a set of physical blocks in Open-Channel SSD, each vblk has a predefined bandwidth. This bandwidth depends on how many LUNs the blocks that make up the vblk exist. For example, if a vblk is configured with physical blocks from all LUNs, this vblk has the maximum bandwidth. In our implementation, IsoKV 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 an 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 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 the main points of the paper.

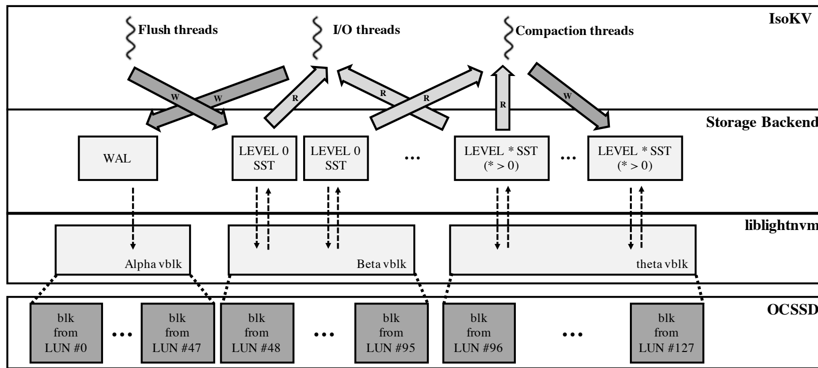
As a result, isolated LUNs divided by a 3:3:2 ratio has improved overall throughput performance over a wide range of workloads with read percentages ranging from 10 percent to 80 percent and dramatically reduced read latency under the workloads with 10%-90% read percentages.

3.4 Dynamic Arrangement of NAND-Flash Parallelism

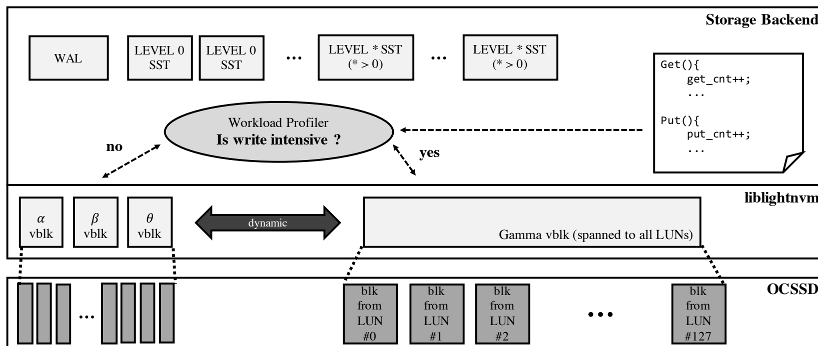
We divide an entire of parallelism (i.e., 128) into 3 portions and assigned each of them to the vblks depending on different needs of parallelism for corresponding their I/O types. As a result, IsoKV with Isolation-Arrangement scheme reduces the delay in read requests due to the long turnaround time of the write request under the mixed reads and writes workload. However, the statically isolated arrangement of LUNs can lead to performance degradation if the nature of the workload changes to extreme write-intensive which read and write do not mix frequently because of the reduced degree of parallelism. To address this kind of problem, we put a simple workload profiler in the IsoKV storage back end to monitor the pattern of workloads. 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 certain period of an interval to determine whether the workload is read intensive or write intensive. If the workload is determined to be write-intensive, the profiler stores the data using gamma vblk rather than alpha, beta, and theta vblk for isolation. This uncomplicated profiler has almost no overhead because there are no



(a) Striping-Arrangement



(b) Isolation-Arrangement



(c) Workload profiler

Figure 3.2: IsoKV storage backend scheme using Open-Channel SSD and liblightnvm

additional calculations. Figure 3.2c shows how the profiler works, and IsoKV decides which arrangement parallelism to use to store the data by this profiler. We evaluated the performance of the Dynamic-Arrangement scheme of IsoKV under changing workload of read: write ratio on the runtime comparing with Striping-Arrangement scheme and Isolation-Arrangement scheme.

In summary, we implemented IsoKV that features an application-driven flash management scheme considering the behavior and data structure of LSM-tree based key-value store. In IsoKV, 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 workloads. The main benefits of IsoKV include: 1) each type of write requests has their own region, thereby alleviating the delay of read comes from the expensive turnaround time of write requests; 2) It shows predictable performance due to reduced interference between I/O types and garbage collection (GC); 3) It uses the optimal device parallelism according to the I/O pattern of the workload.

3.5 Implementation

To verify the efficiency of IsoKV's design strategies, we implement IsoKV based on RocksDB [13], an LSM-tree based KV store for fast storage. We use the liblightnvm [7] interface to access a 2TB Open-Channel SSD from CNEX. liblightnvm allows user space applications to implement direct access to Open-Channel SSD via C API such as *nvm_dev_open, nvm_vblk_pread, nvm_vblk_write, etc.* [14], facilitating application development.

As shown in Figure 3.2, IsoKV maintains virtual blocks (vblks) on the storage back end. I/O generated according to the LSM-tree logic are handled by

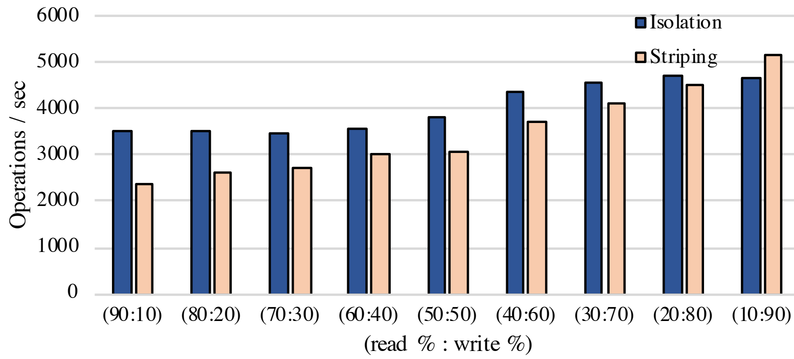
using the vblk. We Implemented the storage back end of IsoKV based on [26]. Specifically, we implement IsoKV's storage back end to: 1) set maps between vblks to physical blocks with configured bandwidth(e.g., 48 out of 128 LUNs); 2) read/write each file through corresponding vblks considering I/O types and level of SST files; 3) erase vblk(corresponding physical blocks) and release files when they are outdated according to the LSM-tree logic; 4) manage file metadata for recovery because IsoKV's I/O bypass existing filesystem; 5) manage weariness of NAND Flash blocks; 6) manage bad blocks in the granularity of vblk; 7) profile temporal tendency of reads and writes to determine the arrangement of parallel units.

Chapter 4

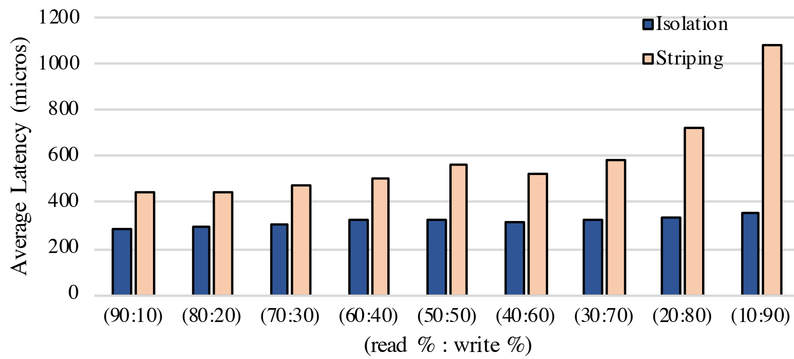
Evaluation

4.1 Experimental Setup

we conducted extensive experiments to evaluate IsoKV 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 an 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 17.04 server and Linux kernel 4.15.0 version for Open-Channel SSD [11] supported IsoKV. We implemented IsoKV based on RocksDB's modified version using Open-Channel SSD and liblightnvm [12].



(a) Overall Throughput (ops/s)



(b) Average Read Latency (micros)

Figure 4.1: Striping-Arrangement VS Isolation-Arrangement.

4.2 Performance Evaluation

In this section, we evaluate the throughput and latency performance of IsoKV using db_bench micro-benchmark released with RocksDB [9]. 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 reads and writes requests, we evaluated the performance of IsoKV under various workloads by changing the read-percentage parameter of db_bench from 10 percent to 90 percent.

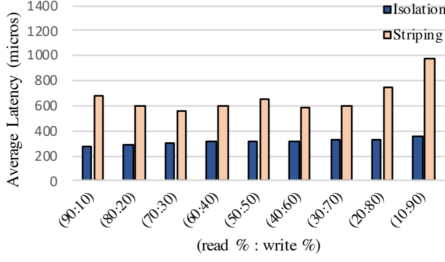
Figure 4.1a plots the IsoKV's overall throughput performance under 9 different workloads which have different percentages of read operation comparing Striping-Arrangement scheme and Isolation-Arrangement scheme of IsoKV. Except for a workload with a read percentage of 10 percent, our work, Isolation-Arrangement 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 occupies 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 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 4.1b represents the average response time results for all read operations. In the Striping-Arrangement scheme, the latency in-

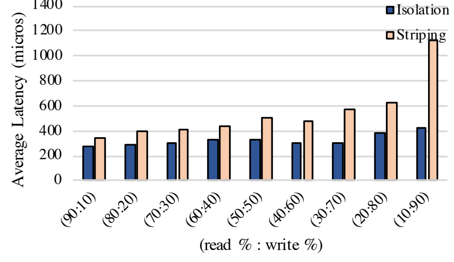
creases proportionally as the ratio of write requests to workload increases. On the other hand, the Isolation-Arrangement scheme always shows predictable performance regardless of the percentage of write requests in the workload. It also achieves 43%-reduced average latency performance result and 67% 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 a detailed analysis of read latency under the 9 workload. In this experiment, read requests reached over 4-levels of SST files from level 0 to level 3. Figure 4.2 and Figure 4.3 illustrate the result of average latency and P99th tail latency. For all workloads with a read percentage of 10 percent to 90 percent, the Isolation-Arrangement scheme reduces average read latency and tail latency at all levels versus the Striping-Arrangement scheme by up to 72% and 96% respectively. Also, In all experiments, the Isolation-Arrangement scheme shows predictable read latency.

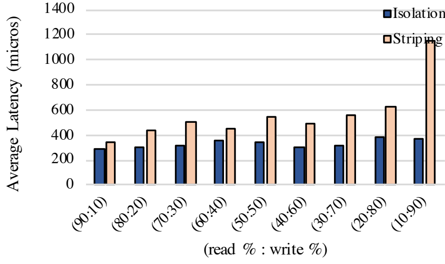
In order to verify the behavior of the dynamic LUNs arrangement scheme, we evaluated IsoKV under workload which changes its characteristics during run-time. The entire run-time of the experiment is divided into 50 intervals, and the interval throughput performance is plotted for each interval. During the first 30 intervals, the workload is read-intensive which has 10% of read-percentage, and the remaining 20 intervals, IsoKV process write-intensive workloads with 90% of read operations. the experimental result shows that the performance of the Isolation scheme represents the 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 a write-intensive workload after the 30th interval shows performance degradation due to degraded parallelism rather



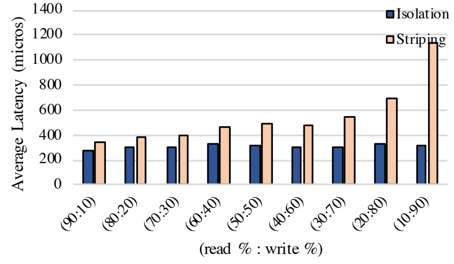
(a) Level-0 SST file



(b) Level-1 SST file



(c) Level-2 SST file

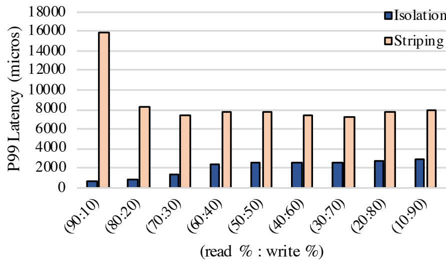


(d) Level-3 SST file

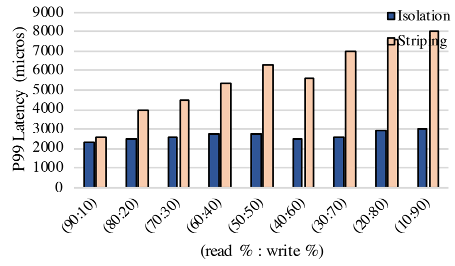
Figure 4.2: Average Read Latency of key-value pair from Level-0 to Level-3 SST files

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.

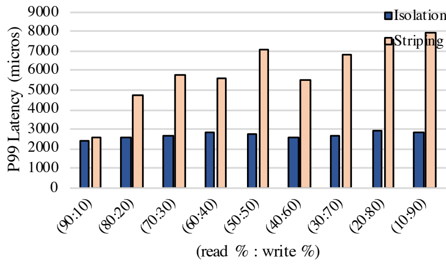
To evaluate IsoKV's performance with more realistic workloads, we run the YCSB benchmark [40] on the Isolation scheme and Striping scheme respectively. The YCSB benchmark is an industry-standard macro-benchmark which has multiple workloads with parameters shown in Table 4.1. Figure 4.5 illustrates the result of four representative workloads(i.e., A, B, D, F). As shown in Figure 4.5 and Figure 4.5b, overall throughput performances and latency are improved compared to the Striping-Arrangement scheme under all workloads.



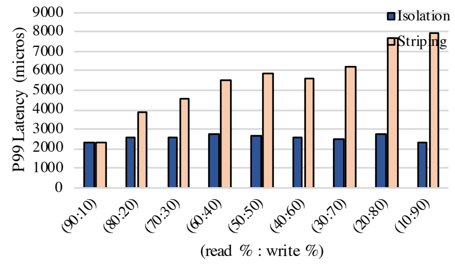
(a) Level-0 SST file



(b) Level-1 SST file



(c) Level-2 SST file



(d) Level-3 SST file

Figure 4.3: P99th Tail Latency of key-value pair from Level-0 to Level-3

Especially under the YCSB-A, an update heavy workload, average latency reduced by 60%. There were also 57% and 53% throughput improvements respectively in YCSB-D and YCSB-F which insert additional records. This is because the I/O interference reduction effect of the Isolation-Arrangement scheme is more pronounced in these insert worklosds

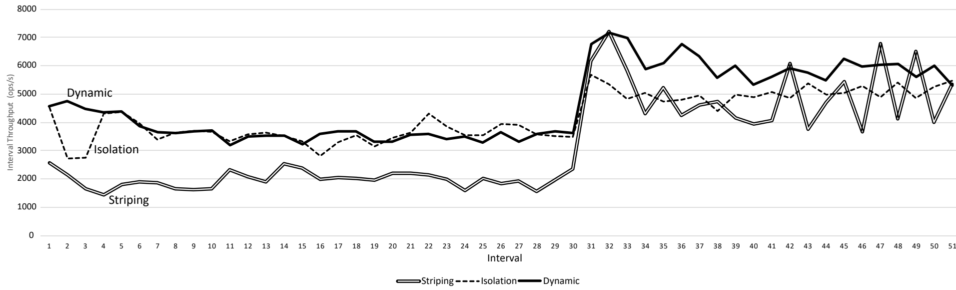
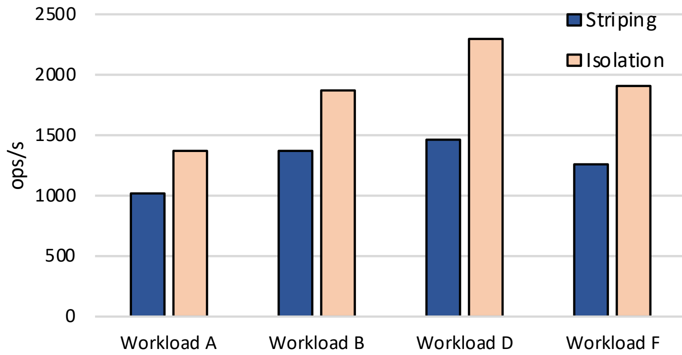


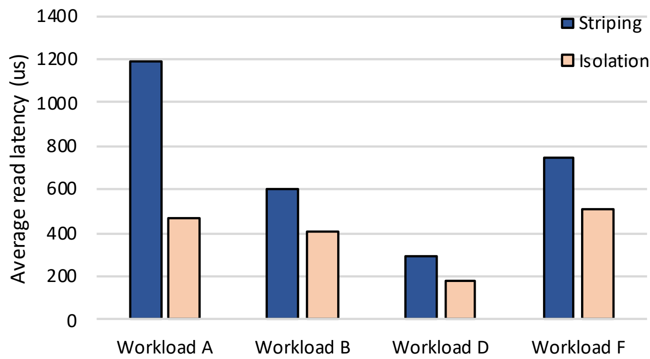
Figure 4.4: Interval throughput comparison under changing workload - Striping VS Isolation VS Dynamic

Workloads	Descriptions	Parameters
YCSB-A	Update heavy workload	readproportion=0.5, updateproportion=0.5
YCSB-B	Read mostly workload	readproportion=0.95, updateproportion=0.05
YCSB-D	New records are inserted	readproportion=0.95, insertproportion=0.05
YCSB-F	Short ranges of key	readproportion=0.5, readmodifywriteproportion=0.5

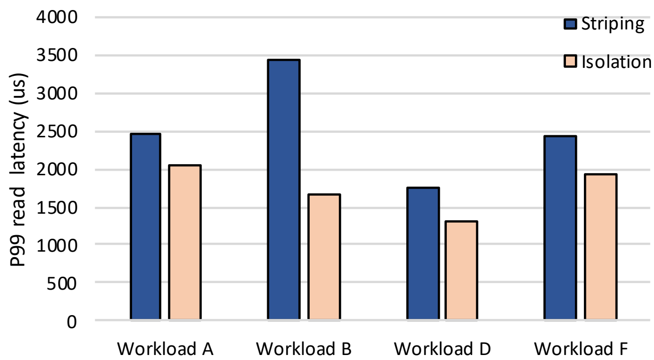
Table 4.1: Workload descriptions and parameters



(a) Overall Throughput (ops/s)



(b) Average Read Latency (micros)



(c) P99 Read Latency (micros)

Figure 4.5: YCSB Macro-benchmarks result

Chapter 5

Related Work

Characteristics of NAND flash based storage device. The previous studies have studied the performance and characteristics of SSDs. The work in [22] presents a set of experiments on the effect of reads/writes and access patterns on performance, while uFlip [20] [21] represents benchmark results and illustrate performance pattern of SSDs and Open-Channel SSDs. In addition, authors in [19] study the effect of parallelism on performance. The work in [33], [34] shows that the randomness of the workload increases garbage collection overhead of the SSD and internal fragmentation, resulting in performance degradation. Even though internal FTL optimizations can mitigate the overhead of random write requests, performance improvement as much as sequential performance is difficult to achieve due to the limited capacity of write buffer inside SSDs [32, 20, 31]. Our works inspired by these works [22, 20, 21, 19, 33, 32, 34, 31]. In contrast, we focus on performance degradation resulting from the interference between reads and writes.

Providing QoS through isolated I/O operations. Rail [24] proposes an

approach based on redundancy to resolve read latency performance degradation of DBMS under read/write workloads. Rail achieves predictable read response time by using additional storage devices that periodically synchronizing each other and having a copy of original data. Under a read/write mixed workload, Rail physically handles either read or write requests per SSD. Also, Woong et al. [38] designed and implemented host side storage engine which schedules I/O and SSD internal operations to data blocks replicated among multiple SSDs. Under the storage engine, latency heavy operations such as garbage collection are detained in a group of SSDs, while foreground I/O requests are delivered to other SSDs. These approaches eliminate the interference between read and write requests by physically isolating the requests. Our study is inspired by these works [24, 38] in terms of providing QoS through isolated requests on each processing unit (SSD, Channel, die, etc.). In contrast, our work fully uses the storage device's capacity, while previous schemes based on redundancy do not fully utilize the capacity.

Cross-layer optimization. In the multi-stream [25], 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. Especially, PCStream [23] extracts program contexts during runtime and automates data-to-stream mapping considering the lifetime of data. Experimental result shows that PCStream reduces the average Write Amplication Factor (WAF) by 35% over existing scheme. FlashShare [28] satisfies different levels of I/O service latency requirements for different co-running applications. It reduces I/O interference among co-running applications bypassing attributes of applications through all the layers of underlying storage stack spanning from Linux kernel to storage devices. For given attributes, each layer of storage stack manages I/O depending on the application type(latency-critical and not). Our

study is in line with these works [25, 23, 28] in terms of that multiple layers of storage stack were co-designed and optimized. In contrast, we focus not on delivering the hints of the host software to the hardware, but on the host software directly managing the flash-based storage devices.

Managing storage devices at the host layer. [30] aims to solve the double logging problem in both FTL and append-only application. Also, GearDB [29] shares the consideration of separating data with similar lifetimes. Javier et al. [26] present application-driven FTL that eliminates redundant logic between application and SSD's internal FTL. Likewise, SDF [39] exposes SSD's internal flash channels to the host software and eliminates space over-provisioning. The host software, given direct access to the raw flash channels of the SSD, effectively organize its data and schedule its data access. Our study is in line with these works [30, 29, 26, 39] in terms of managing storage devicese at the host layer. Especially, our work is inspired by [26, 29] in terms of that they eliminate on-storage garbage collections by synchronizing application logic and behavior of storage devices. In contrast, we enable flash management such as data placement, garbage collection (GC) considering the interference between I/O components in an application.

Chapter 6

Conclusion

In this paper, we present IsoKV, an LSM-tree based key-value store tailored for Open-Channel SSDs.IsoKV achieves both enhanced performance of throughput and latency with three main design approaches: GC-free flash management, isolated data placement policy, and dynamic arrangement of parallelism. We implement IsoKV on RocksDB and evaluate it with a real Open-Channel SSD. Experimental results show that IsoKV improves the overall performance by about 1.20× under various micro-benchmark workloads compared with the existing scheme. Also, most importantly, IsoKV reduces average and tail read latency by up to 67% and 96%, respectively. Furthermore, under the realistic workloads, IsoKV achieves improves the overall throughput by 1.36× and reduces the average read latency by 60% compared with the existing scheme. The performance gain mainly comes from eliminated interference between I/O types and dynamic arrangement of parallelism based on workload characteristics.

Bibliography

[1] FIO: Flexible I/O tester, https://linux.die.net/man/1/fio

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

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

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

[5] Google LevelDB, https://github.com/google/leveldb

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

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

[8] oc_bench: benchmark tool for Open-Channel SSDs, https://github.com/RockyLim92/ocssd_bench

[9] rocksdb db_bench tool, https://github.com/facebook/rocksdb/wiki/Benchmarking-tools

[10] SAMSUNG 960 PRO SSD Specification, https://www.samsung.com/semiconductor/minisite/ssd/product/

[11] OpenChannelSSD/linux repository,
https://github.com/OpenChannelSSD/linux/tree/pblk.cnex

[12] RockyLim92/rocksdb repository,
https://github.com/RockyLim92/rocksdb

[13] OpenChannelSSD/rocksdb repository,
https://github.com/OpenChannelSSD/rocksdb

[14] liblightnvm C API,
http://lightnvm.io/liblightnvm/capi/index.html

[15] 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.

[16] 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.

[17] Heerak Lim. "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.

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

[19] 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.

[20] Bouganim, Luc, Björn Jónsson, and Philippe Bonnet. ”uFLIP: Understanding flash IO patterns.” arXiv preprint arXiv:0909.1780 (2009).

[21] Picoli, Ivan Luiz, et al. ”uFLIP-OC: Understanding flash I/O patterns on open-channel solid-state drives.” Proceedings of the 8th Asia-Pacific Workshop on Systems. ACM, 2017.

[22] Chen, Feng, David A. Koufaty, and Xiaodong Zhang. ”Understanding intrinsic characteristics and system implications of flash memory based solid state drives.” ACM SIGMETRICS Performance Evaluation Review. Vol. 37. No. 1. ACM, 2009.

[23] Kim, Taejin, et al. ”PCStream: automatic stream allocation using program contexts.” 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18). 2018.

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

[25] 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.

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

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

[28] Zhang, Jie, et al. "Flashshare: punching through server storage stack from kernel to firmware for ultra-low latency SSDs." 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). 2018.

[29] Yao, Ting, et al. "GearDB: A GC-free Key-Value Store on HM-SMR Drives with Gear Compaction." 17th USENIX Conference on File and Storage Technologies (FAST 19). 2019.

[30] Lee, Sungjin, et al. "Application-managed flash." 14th USENIX Conference on File and Storage Technologies (FAST 16). 2016.

[31] Zhou, You, et al. "An efficient page-level FTL to optimize address translation in flash memory." Proceedings of the Tenth European Conference on Computer Systems. ACM, 2015.

[32] Chen, Feng, David A. Koufaty, and Xiaodong Zhang. "Understanding intrinsic characteristics and system implications of flash memory based solid state drives." ACM SIGMETRICS Performance Evaluation Review. Vol. 37. No. 1. ACM, 2009.

[33] Min, Changwoo, et al. "SFS: random write considered harmful in solid state drives." FAST. Vol. 12. 2012.

[34] Mittal, Sparsh, and Jeffrey S. Vetter. "A survey of software techniques for using non-volatile memories for storage and main memory systems." IEEE Transactions on Parallel and Distributed Systems 27.5 (2015): 1537-1550.

[35] Hu, Xiao-Yu, et al. "Write amplification analysis in flash-based solid state drives." Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference. ACM, 2009.

[36] Lee, Sang-Won, et al. "A log buffer-based flash translation layer using fully-associative sector translation." ACM Transactions on Embedded Computing Systems (TECS) 6.3 (2007): 18.

[37] Lee, Sungjin, et al. "LAST: locality-aware sector translation for NAND flash memory-based storage systems." ACM SIGOPS Operating Systems Review 42.6 (2008): 36-42.

[38] Shin, Woong, et al. "Providing QoS through host controlled flash SSD garbage collection and multiple SSDs." 2015 International Conference on Big Data and Smart Computing (BIGCOMP). IEEE, 2015.

[39] Ouyang, Jian, et al. "SDF: software-defined flash for web-scale internet storage systems." ACM SIGPLAN Notices. Vol. 49. No. 4. ACM, 2014.

[40] Cooper, Brian F., et al. "Benchmarking cloud serving systems with YCSB." Proceedings of the 1st ACM symposium on Cloud computing. ACM, 2010.

[41] Bjørling, Matias, Javier González, and Philippe Bonnet. "LightNVM: The Linux Open-Channel SSD Subsystem." 15th USENIX Conference on File and Storage Technologies (FAST 17). 2017.

[42] Bjørling, Matias, et al. "Linux kernel abstractions for open-channel solid state drives." Non-Volatile Memories Workshop. 2015.

초록

최신 데이터 센터는 스토리지 서버, 캐시 시스템 및 Key-Value stores와 같은 I/O 집약적인 애플리케이션을 위한 스토리지 장치의 높은 병렬성을 활용하는 것을 목표로 한다. Key-value stores는 고성능의 고신뢰 서비스를 제공해야 하는 가장 대표적인 응용프로그램이다. Key-value stores의 I/O 성능을 높이기 위해 많은 데이터 센터가 비휘발성 메모리 익스프레스(NVMe) 기반 SSD(Solid State Devices)와 같은 차세대 스토리지 장치를 적극적으로 채택하고 있다. NVMe SSD와 그 프로토콜은 높은 수준의 병렬성을 제공하는 것이 특징이다. 그러나 NVMe SSD가 병렬성을 제공하면서도 예측 가능한 성능을 보장하지는 못할 수 있다. 예를 들어 읽기 및 쓰기 요청이 많이 혼합되면 요청과 내부 작업(예: GC) 사이의 간섭으로 인해 처리량 및 응답 시간의 성능 저하가 발생할 수 있다.

간섭을 최소화하고 성능을 향상시키기 위해 본 연구에서는 Key-value stores를 위한 격리 방식인 IsoKV를 제시한다. IsoKV는 애플리케이션 중심 플래시 저장장치 관리 방식을 통해 SSD의 병렬화 수준을 직접 관리한다. IsoKV는 SSD의 각 전용 내부 병렬 장치에 서로 다른 특성을 가진 데이터를 저장함으로써 I/O 요청 간의 간섭을 줄인다. 또한 IsoKV는 SSD의 LSM 트리 로직과 데이터 관리를 동기화하여 GC를 제거한다. 본 연구에서는 RocksDB를 기반으로 IsoKV를 구현하였으며, Open-Channel SSD를 사용하여 성능평가하였다.. 본 연구의 실험 결과에 따르면 IsoKV는 기존의 데이터 저장 방식과 비교하여 평균 1.20× 빠르고 및 43% 감소된 처리량과 응답시간 성능 개선 결과를 얻었다. 관점에서 43% 감소하였다.

주요어: Storage, NAND-flash, Open-Channel-SSD, FTL, NVMe, LSM-tree
학번: 2017-21118