Improving Performance in LSM-Tree based Key-Value Stores using NVMe

Satoshi Iwata Mohil Patel Partho Sarthi John Shawger
University of Wisconsin-Madison

ABSTRACT

Log-Structured Merge-Tree key-value stores convert random key value insertions into sequential writes on disk. These systems historically perform well on spinning hard disks, but their log-structured nature leads to high write amplification. How to best utilize newer and faster storage technologies like NVMe with these systems remains an open question. We evaluate RocksDB performance on machines with SATA SSD and NVMe SSD disks, and propose a method to maximize overall system throughput by writing to both disks concurrently.

1 INTRODUCTION

RocksDB[2] is a persistent key-value store for fast storage environments. It is based on the technology of the Log-Structured Merge-Tree[14] (LSM-Tree) and provides high insert performance. It is originally developed by Facebook based on LevelDB and has been adapted to various systems. Use cases include a number of backend systems at Facebook and storage engines of popular databases such as MySQL and MongoDB.

The NVM Express (NVMe) is an interface developed to fully utilize performance of non-volatile memories such as NAND flash memory. Though conventional Serial AT Attachment (SATA) interface has been initially utilized for accessing Solid-State Drives (SSDs), SATA is originally developed for Hard Disk Drives (HDDs) and it restricts SSDs from providing their maximum performance.

With cost differences between NVMe SSD and SATA SSD in mind, an interesting approach has been proposed by SpanDB[8]. SpanDB is based on RocksDB and it utilizes a combination of NVMe SSD and SATA SSD. It stores hot Write-Ahead Log (WAL) and top levels of LSM-Tree into an expensive NVMe SSD and stores cold bottom levels of LSM-Tree into a cheaper SATA SSD. By doing so, SpanDB is able to provide high performance with limited affordable cost.

Motivated by SpanDB, we propose a method which splits hot data[1] to NVMe SSD and SATA SSD and provides the sum performance of both devices. Though performance of SATA SSD is relatively lower compared to NVMe SSD, SATA SSD is enough fast and we believe it is waste of the device performance to store only cold data in it. By splitting hot data to NVMe SSD and SATA SSD and utilizing the performance of both disks, we try to make RocksDB even faster compared to SpanDB's approach.

We implemented our approach in two ways to see the performance differences. One is a method which splits data in Sorted String Table (SSTable) file granularity. The other is a method which splits data in block granularity using software Redundant Array of Independent Disks (RAID).

We conducted some experiments to show the performance improvement by our approach. In contrast to our expectations, our approach falls between the performance using single NVMe SSD and performance using single SATA SSD. Though we tried to discover the root cause of the performance bottleneck in our experiments, unfortunately, we could not discover it due to time frame limitations.

The rest of this paper is organized as follows. Section 2 explains RocksDB and evolution of storage technologies in more detail as a background of our research. After showing our current design and implementations in Section 3, we illustrate current experimental results in Section 4. We review some related works in Section 5 and conclude our paper in Section 6 with pointing some future directions.

[1]Our current implementation splits whole LSM-Tree, instead of splitting only hot top levels of LSM-Tree. Due to some reasons for experimentations, our current implementation does not utilize WAL.

2 BACKGROUND

2.1 RocksDB

RocksDB [2] is an open-source, embedded key-value store designed to be fast and efficient. It is implemented using a log-structured merge-tree (LSM [14] tree) data structure, which allows it to store and retrieve data quickly, even when dealing with large volumes of data. An LSM tree is a data structure used to store data on disk or other persistent storage. It consists of two or more levels of storage: a fast, in-memory level that is used for recent writes, and one or more slower, persistent levels that are used for older data. When data is written to the in-memory level, it is eventually merged with the persistent levels in a process called compaction. This helps keep the LSM tree balanced and allows it to remain efficient as data grows. LSM trees are well-suited for applications that require high write performance, such as databases and messaging systems, but can have higher read latencies and require additional space to store the in-memory and persistent levels of data.

In the RocksDB write path, as shown in Figure 1, any key-value pair write request made by the user is first written to a Write Ahead Log (WAL) for persistence. The key-value pair is then written to an in-memory data structure called a memtable. A memtable is essentially a buffer that stores new data in memory until it can be written to disk. RocksDB stores multiple memtables in memory and switches between them as they fill up. In the background, filled-up memtables are pushed to disk by a background flush thread, which converts the memtables into SSTables (Sort String Tables) before pushing them to level 0. An SSTable is a file on disk that stores a sorted list of key-value pairs. Once level 0 fills up, another background thread comes in to do compaction. The compaction process reads the SSTables from a level, removes duplicates, and collates the data into bigger SSTables before pushing them to a lower level. This process helps to reduce the number of stale and duplicate entries in the database.

In the read path of RocksDB (as shown in Figure 2), client requests for a specific key-value pair are first checked in the memtables and in-memory cache. If the key is found, the value is returned immediately. If the key is not found in the in-memory data structures, RocksDB goes to the disk to search for the value. While searching the levels on the disk, RocksDB starts from the lower levels and moves toward the higher levels to avoid stale

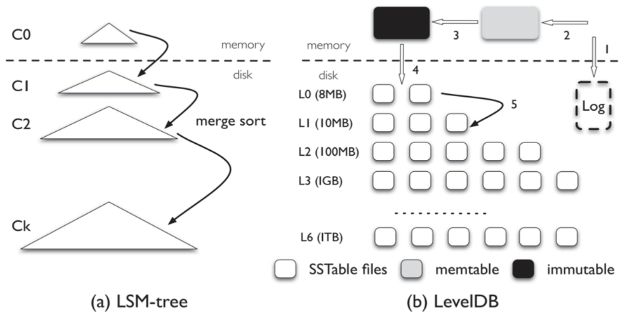


Figure 1: RocksDB Write Path [13]

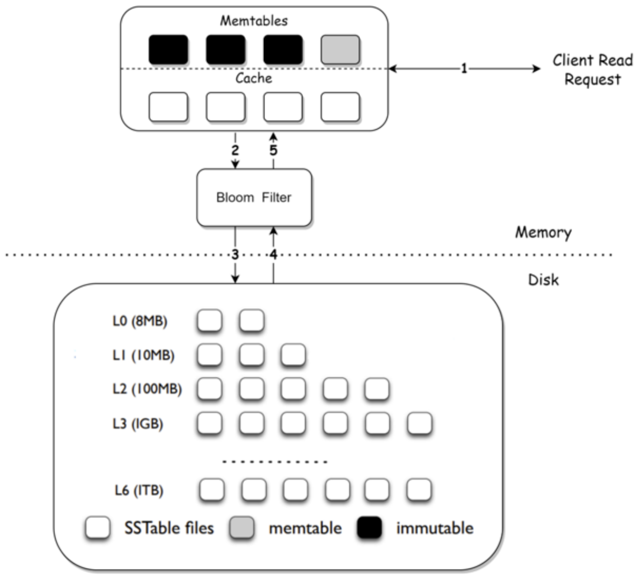


Figure 2: RocksDB Read Path

or duplicate entries that may exist across levels. To improve read performance, RocksDB uses a per SSTable Bloom filter [7] to quickly identify whether the SSTable needs to be read or skipped. This check helps to reduce the number of disk reads and improve performance.

2.2 SATA, NVMe, and SPDK

SATA is an interface originally developed to connect devices such as HDDs and optical devices to computers. So, its maximum bandwidth is 600 MB/s and cannot fully utilize performance of SSDs.

NVMe has been developed to fully utilize performance of SSDs. It has various characteristics to improve performance. One example is increased number of queue depth to provide high parallelism. While SATA

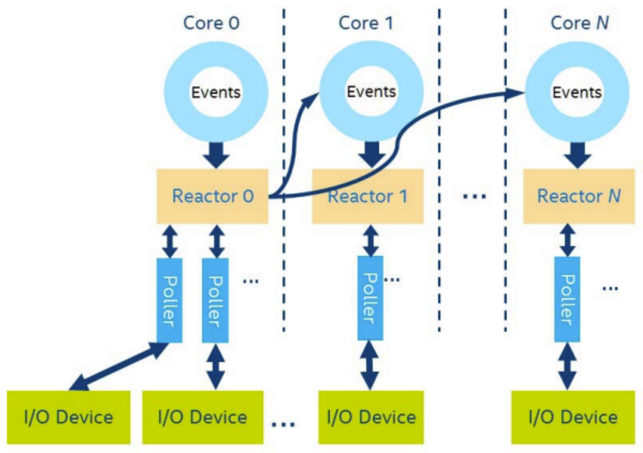


Figure 3: SPDK Application Event Framework [18]

provides one command queue which holds up to 32 commands, NVMe provides 65535 command queues which holds up to 65536 commands per queue. By implementing these optimizations, NVMe provides maximum of 16,000 MB/s, which is 26 times higher compared to SATA interface.

Another important technology is Storage Performance Development Kit[18] (SPDK). It is a library which provides high-speed storage access based on two main characteristics. One is user-mode storage access to avoid context switches between applications and operating systems. The other chracteristic is polling-based completion check to avoid interrupt overhead. SPDK event framework uses event queues and polls the I/O device to check for completion, as shown in Figure 3. It uses a lockless architecture and a user-space driver to improve performance. Same as SpanDB, our implementation uses SPDK to utilize the maximum performance of NVMe SSDs.

We conducted some measurements using `fio`[1] benchmarking tool to check performances differences between these technologies. Figure 4 shows bandwidth and Figure 5 shows IOPS. We set block size to 1MB and 4KB respectively for measurement of bandwidth and IOPS. To measure the performance improvement with SPDK, we also measure performance with ext4 file system.

Figure 4 and Figure 5 illustrates that ext4 file system over NVMe SSD provides 4-5 times performance improvement in bandwidth and more than 3 times performance improvement in IOPS respectively compared

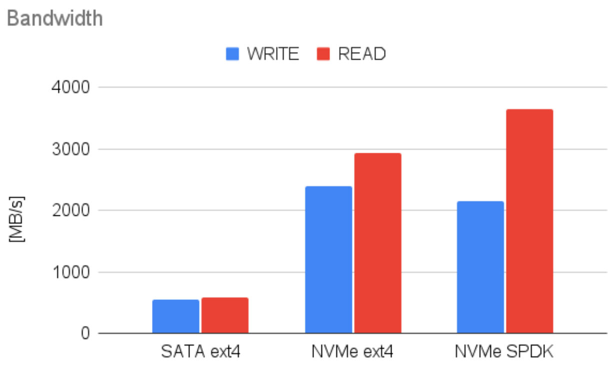


Figure 4: Bandwidth measured with `fio`

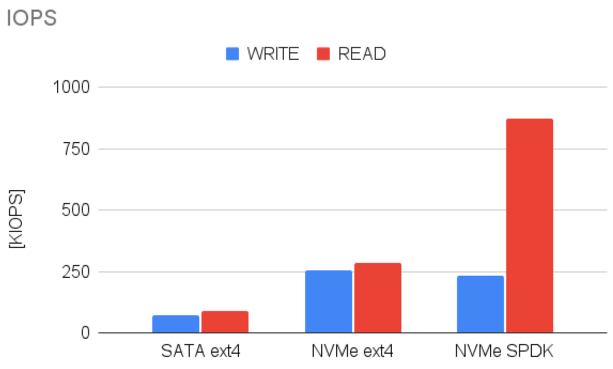


Figure 5: IOPS measured with `fio`

to ext4 file system over SATA SSD in both READ and WRITE operations.

SPDK provides higher performance for READ operations over ext4 file systems. Especially, READ IOPS increased about three times. WRITE operation has already been saturated with ext4 file system, so SPDK did not provide further performance improvement. These results are consistent with specifications provided by vendors[3, 4] except for READ bandwidth. We could not discover the reason of the difference.

3 DESIGN AND IMPLEMENTATION

Motivated by the knowledge that background compaction consumes a great deal of disk bandwidth, we explored ways to maximize the disk throughput of our system. Other recent work, such as SpanDB [8], use placement strategies to take advantage of the lower latency achievable with NVMe and SPDK. For example, since write-ahead logging is on the critical write path, lower latency times in write-ahead logging significantly impact system performance. Since this type of strategy had already been explored, we chose to disable write-ahead logging and maximize the overall sequential write throughput of the system, with the hope that higher throughput during compaction would lead to better system performance in write-heavy workloads.

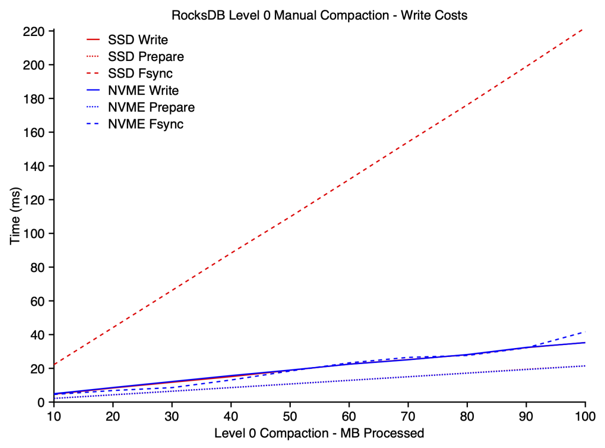


Figure 6: RocksDB Compaction Write Time

In order to maximize disk bandwidth, we wanted to use both disks in parallel. While the NVMe SSD is much faster than the SATA SSD, leaving the SATA SSD unused while writing to the NVMe device would limit the overall bandwidth of the system. To determine how expensive disk writes during compaction are, we performed an experiment in which level 0 was filled with between 10-100MB of sequentially-keyed data (Figure 6). Default compaction was turned off, and we manually applied a simple compaction algorithm which read in level 0 files and wrote them out in level 1. This experiment was done on both disks. `Write` and `fallocate` system calls, which are only dependant on the operating system for their operation, return in the same amount of time regardless of which disk they are performed on. `Fsync` system calls show a large performance difference depending on the disk. Since sync and flush are disabled in ordinary compaction, we determined that we could issue writes to both disks without one operation blocking the other.

We used two strategies to split data between disks. SSTable files are named by their number – `000012.sst`, for example. By using a modulo operator, we were able to store SSTable files on either disk in 1:1, 1:3, and 1:5 ratios, with the larger portion stored on the NVMe disk. Separately, we used `mdadm`, which is a software RAID tool included with Linux to stripe data between the disks.

4 EVALUATION

We evaluated our system on a CloudLab machine with 64 Cores (AMD 7543) and 256GB Memory. It contained two SSDs connected using a SATA and an NVMe link. In this section, we describe the benchmarking tool used for the experiments followed by a discussion of the results.

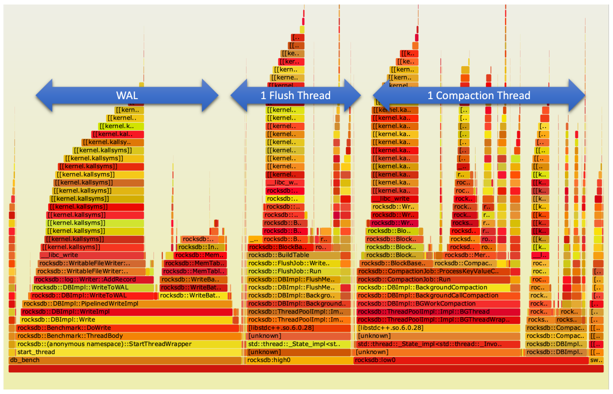
4.1 Benchmarking Tool

db_bench [10] is the main tool used for measuring the performance of our system. It is developed by the RocksDB team with some modifications from the LevelDB version. It supports various configurations such as the number of background jobs, workload categories (random and sequential reads and writes), compression, and toggling write-ahead-logging. After running some initial benchmarks with default configuration, we made the following observations: (1) Sequential writes did not trigger compaction. (2) WAL without sync did not show device performance (instead just the cache performance). (3) Default size of the database was not large enough to incur compaction. Thus, we selected the following workload - Random writes with a database size of 20 GB.

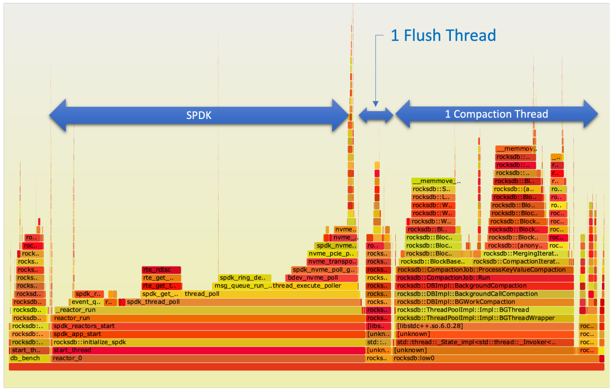
As mentioned previously, RocksDB runs the flush and compaction jobs in the background. The number of threads for these background jobs can be configured in *db_bench* either by specifying the number of flush and compactions threads explicitly or by specifying the total number of jobs. RocksDB calculates these values using the following equation:

$$Jobs_{flush} = \frac{Jobs_{total} + 3}{4} \tag{1}$$

$$Jobs_{comp} = Jobs_{total} - Jobs_{flush} \tag{2}$$



(a) Running on ext4



(b) Running on SPDK with WAL disabled

Figure 7: Flamegraph of *db_bench*

4.2 Results

Figure 7a shows a flamegraph of running *db_bench* on an ext4 filesystem. There are majorly three mountains denoting three parallel jobs - min thread running *db_bench* , background flush and background compaction jobs. From the main thread, we observed that WAL consumes a significant number of CPU cycles. This would be a bottleneck for our experiments and the performance differences would not show up correctly. Hence, we disabled WAL for all our experiments.

Figure 7b shows a flamegraph of running *db_bench* on SPDK with WAL disabled. We observed that the number of cycles for background flush jobs reduced from 21% to 4%. This is because the actual writes are now being handled by the SPDK thread. The flush job posts as async request to the SPDK thread, which performs the actual write to the NVMe SSD.

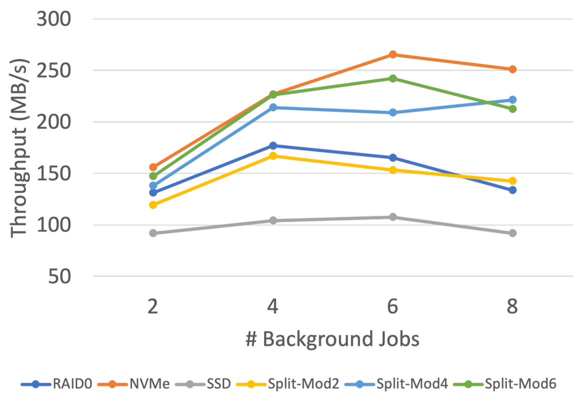
We evaluated our splitting-based mechanism on two filesystems. Figure 8a and 8b show the throughput and latency of running RocksDB on ext4 by varying the number of background jobs. We measured the performance of NVMe, SATA SSD and the splitting techniques for various modulo values. For example, Split-Mod4 refers to 3 parts in NVMe and 1 part in SATA SSD. We also evaluated RAID0 which strips the data at a block-level granularity and divides them across multiple devices simultaneously. The results show that NVMe is the fastest followed by Split-Mod6, Split-Mod4 and so on, SATA SSD being the slowest. We also observed the similarity in the performance of RAID0 and Split-Mod2.

Next, we performed a similar experiment on SPDK and compared it with NVMe and SATA SSD. Figure 8c and 8d show the throughput and latency of RocksDB on SPDK. Since we expect a caching layer in NVMe (which is absent in SPDK), we enabled direct I/O for NVMe writes in this case. Splitting results are similar to the ext4 case. However, the key observation is that the performance of SPDK-Mod4 is better than the performance of vanilla SPDK. This correlates to our proposal that using the splitting method we are able to utilize the bandwidth of both devices simultaneously.

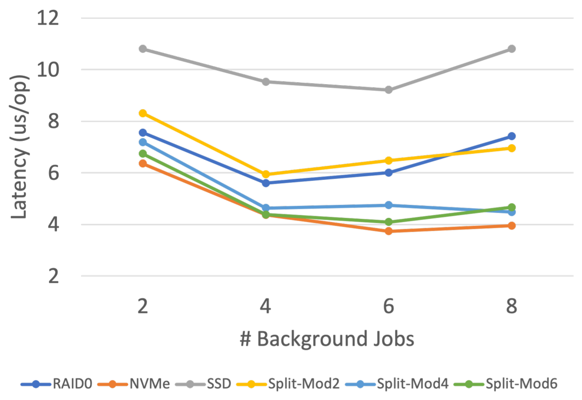
5 RELATED WORKS

The developers of RocksDB [9] recognize that the design goals of the system have changed over time due to hardware trends. RocksDB is very tunable, which allows it to adapt to different system environments. The question of how to best use RocksDB with Storage Class Memory and other new storage technologies has been explored by many research groups over the past several years.

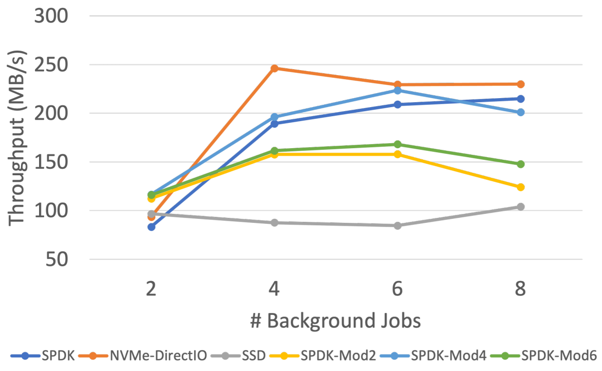
Recent papers focus on reducing latency by adapting I/O scheduling based on the load characteristics from the client [5]. SPDK [18] promises a way to reduce I/O latency via user-space device polling, and avoids modifications to the kernel I/O stack [6] [16]. A recent system, Vigil-KV [12], uses SPDK with NVMe-specific features such as Sets and Predictable Latency Mode (PLM) to improve `Get` operation latency. SpanDB [8] uses SPDK and a novel placement strategy to reduce latency along the critical write path. Unlike these systems, our work focused exclusively on increasing write speed during compaction by utilizing all available disk bandwidth.



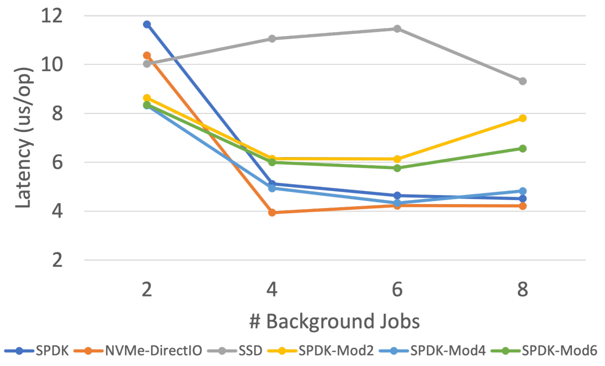
(a) ext4 Throughput



(b) ext4 Latency



(c) SPDK Throughput



(d) SPDK Latency

Figure 8: Performance comparison of RocksDB by varying the number of background jobs

ListDB [11] and Pacman [17] use byte-addressable persistent memory to perform compaction more efficiently. We did not explore uses for persistent memory.

6 CONCLUSION AND FUTURE WORK

Our approach consisted of splitting the LSM tree at a file level (SSTable) granularity. From our experiments, we observed that the performance of this approach was in between NVMe and SSD. However, the splitting technique using SPDK had better performance than using vanilla SPDK. Thus, we were able to utilize the bandwidth of both devices simultaneously. The flamegraphs showed that SPDK uses only a single event queue (reactor) for all reads and writes. We believe that having multiple event queues in SPDK, along with caching, could improve the performance of our system.

For future works, we propose two major directions. First, improve the performance of the system by addressing the limitations of the BlobFS filesystem used in the project. Specifically, implement caching in BlobFS and allow for multiple writer threads. The second is to investigate the possibility of dynamically placing files on disks based on real-time disk bandwidth availability rather than static split. By leveraging the SPDK event framework's event queue (Figure 3) and using the event queue size as a proxy for disk load (a technique similar to Shenango [15]), we can optimize file placement for maximum performance at runtime.

REFERENCES

[1] Flexible I/O Tester. https://github.com/axboe/fio. Accessed: 2022-12-17.

[2] RocksDB. http://rocksdb.org/. Accessed: 2022-12-17.

[3] Specification of Samsung PM1733/PM1735 NVMe Solid state

drive. https://semiconductor.samsung.com/us/ssd/enterprise-ssd/pm1733-pm1735/. Accessed: 2022-12-16.

[4] Specification of Samsung SM883 MZ7KH480HAHQ0D3 SATA Solid state drive. https://www.serversupply.com/SSD/SATA-6GBPS/480GB/SAMSUNG/MZ7KH480HAHQ0D3_317229.htm. Accessed: 2022-12-16.

[5] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona. SILK: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 753–766, 2019.

[6] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. Linux block io: Introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, New York, NY, USA, 2013. Association for Computing Machinery.

[7] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, jul 1970.

[8] H. Chen, C. Ruan, C. Li, X. Ma, and Y. Xu. SpanDB: A fast, cost-effective LSM-tree based KV store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 17–32, 2021.

[9] S. Dong, A. Kryczka, Y. Jin, and M. Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49, 2021.

[10] Facebook. Benchmarking tools · facebook/rocksdb wiki. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[11] W. Kim, C. Park, D. Kim, H. Park, Y.-r. Choi, A. Sussman, and B. Nam. ListDB: Union of Write-Ahead logs and persistent SkipLists for incremental checkpointing on persistent memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 161–177, 2022.

[12] M. Kwon, S. Lee, H. Choi, J. Hwang, and M. Jung. Vigil-KV: Hardware-Software Co-Design to integrate strong latency determinism into Log-Structured merge Key-Value stores. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 755–772, 2022.

[13] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):1–28, 2017.

[14] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.*, 33(4):351–385, jun 1996.

[15] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, 2019.

[16] W. Shin, Q. Chen, M. Oh, H. Eom, and H. Y. Yeom. OS I/O path optimizations for flash solid-state drives. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 483–488, Philadelphia, PA, June 2014. USENIX Association.

[17] J. Wang, Y. Lu, Q. Wang, M. Xie, K. Huang, and J. Shu. Pacman: An efficient compaction approach for Log-Structured Key-Value store on persistent memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 773–788, 2022.

[18] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. SPDK: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017.