MoKE: Modular Key-value Emulator for Realistic Studies on Emerging Storage Devices

Manoj Pravakar Saha
Florida International University
Miami, USA
msaha002@fiu.edu

Danlin Jia
Northeastern University
Boston, Massachusetts
jia.da@northeastern.edu

Janki Bhimani
Florida International University
Miami, Florida
jbhimani@fiu.edu

Ningfang Mi
Northeastern University
Boston, Massachusetts
ningfang@ece.neu.edu

2023 IEEE 16th International Conference on Cloud Computing (CLOUD) | 979-8-3503-0481-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/CLOUD60044.2023.00077

***Abstract*—Key-value stores are widely used as building blocks in today's IT infrastructure for managing and storing large amounts of data. Storage technologies are undergoing continuous innovations to accelerate KV workloads. However, designing high-performance KV or object storage devices is challenging and still needs more research to address the performance bottlenecks of the existing designs. There is a void for an inexpensive and extendable research platform that enables in-depth exploration of the index management components within the KV devices. To fill this void, we design Modular Key-value Emulator (MoKE). MoKE is a software emulator for fostering future full-stack software/hardware KV and object storage device research. MoKE is cheap (software-based emulator), usable with SNIA KV API (supports popular host-device interfaces), extendable (supports internal KV device research), and adaptable (QEMU-based).**

***Index Terms*—Solid State Drive, Key-Value SSD, emulator**

I. Introduction

The explosion of unstructured data has given rise to the Key-Value (KV) and object interfaces and unstructured data stores. However, conventional software KV/object databases confront computational and memory overheads on the operating system (OS) I/O stack. I/O requests need to go through multiple layers, and each layer adds syntax translation complexity and performance cost. Therefore, computational and memory overheads of software KV/object stores [6], [11] have intrigued researchers in industry and academia to explore KV/Object Drives over the years. The proposed designs include Kinetic Drive [8], KAML [4], KV-SSD [2], [6], [9], and KV CSD [12]. While each design is innovative, we have not seen wide-scale adoption of such devices. Despite the low adoption of such devices, KV- and Object-SSD retains their intrigue because of the popularity of the KV and object interfaces, which needs constant efforts from both academia and industry.

Research quests to design efficient KV and Object SSDs have been challenging and expensive. Existing research works have all been implemented either on real devices, accessible only to manufacturers or on costly FPGA-based device prototypes. On top of the direct cost of such devices, the learning curve of working on FPGA-based devices is also very high. It is not easy for most academic research labs to afford such expensive and complex platforms. More importantly, there is a void in terms of open-source software platforms that foster research on the impact of individual components within KV storage devices on the overall I/O stack. We need open-source emulators/simulators that can enable research on the primary bottlenecks, such as host-device interface, application data index management, and integrated RAM caching, within KV-SSDs and similar devices [11].

To fill this void, we present Modular Key-value Emulator (MoKE), based on FEMU (a QEMU-based Flash Emulator [10]), to foster in-depth research on KV-SSDs. MoKE provides a complete overview of the impact of individual internal components on the overall I/O stack.

II. MoKE: Modular Key-value Emulator

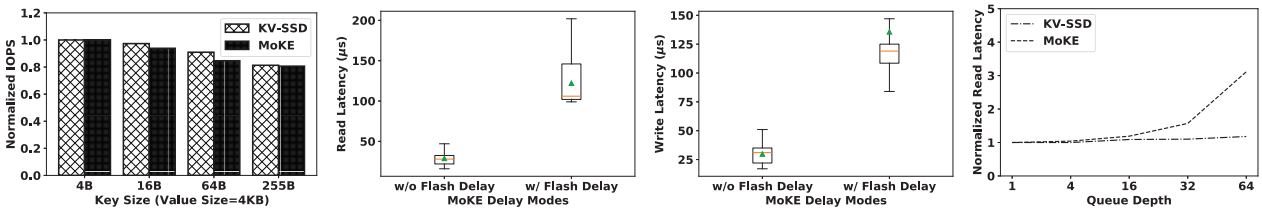
MoKE is designed on top of FEMU [10], a popular block-SSD emulator. Like FEMU, MoKE offers a virtualized SSD interface running inside a virtual machine on top of QEMU [1]. In MoKE, we replace FEMU's block-based host-device interface with NVMe KV interface, with support for the SNIA KV API [15]. We also replace FEMU's data backend and FTL emulation to enable KV data management and delay emulation.

A. Host-Device Interface for KV Stack

The first step in designing an NVMe KV host-device interface is to implement the NVMe KV command set [3] into device drivers and implement KV request handlers inside the emulator. Since, MoKE is a research platform, we integrate MoKE with the more flexible OpenMPDK NVMe KV command set and device driver [14].

1) KV Command Transfer: **Problem:** KV command transfers are inherently different and expensive compared to their block device counterparts and can affect I/O performance of the device [7]. First, keys larger than 16 bytes cannot be passed on as part of a NVMe command and requires a separate DMA transfer using Physical Region Pages (PRP). Secondly, additional packing and unpacking of keys can lead to overhead in command processing. Thirdly, unlike block devices, KV commands cannot be coalesced together based on sequentially or key group. A real Samsung KV-SSD (not the KV emulator) [6] uses multiple request handlers running on different CPUs to offset these overheads. However, implementing emulation of such request-handling architectures would complicate the design and incur significant overhead. It is challenging to correctly emulate the command processing to match real KV-SSD's command processing throughput.

2159-6190/23/$31.00 ©2023 IEEE
DOI 10.1109/CLOUD60044.2023.00077



(a) (b) (c) (d)

Fig. 1: **(a) MoKE IOPS drop with increase in key size. (b) Read I/O latency in different modes. (c) Write I/O latency in different modes. (d) Impact of queue depth on read latency.**

Our Solution: To achieve correct and efficient command processing in MoKE, we first implement separate DMA transfers for keys and values based on command parameters. We also introduce new data structures to identify and process KV commands and to manage the KV data inside the emulator efficiently. Approximately of 900 lines of codes (LOC) has been added in `nvme.h`, `nvme-io.c` and `bb.c` to support the feature of NVMe KV command transfer.

Results: We measure the correctness of the host-device interface by comparing MoKE's throughput trends with Samsung KV-SSD for different key sizes. When the key sizes increase, the I/O throughput of KV-SSD is expected to decrease [13], due to additional DMA transfers. Figure 1a shows that MoKE can correctly emulate this behavior.

2) KV API support: **Problem:** Applications interact with KV-SSD using the standardized SNIA KV API. The KV API library transforms the API requests into NVMe KV commands. For SNIA KV API to work properly, a storage device/emulator needs to implement two features - key space management and KV I/O error handling. However, key space is an abstraction recognized by the KV API, not the NVMe layer. In addition, KV-specific I/O errors during a transaction also needs to be handled by the KV API layer. The exact method through which the API-level abstractions are recognized and processed inside the KV-SSD is not trivial. For example, the data structures that list the key spaces, some details such as the actual keys being used or the exact NVMe commands being used for management of key spaces are still implementation dependent. Hence, to enable keyspace management KV API support, we either need to reverse engineer these implementation details (i.e. keys used to store key space list and commands used to manage key spaces) and modify MoKE, or implement our own solution both in the KV API layer and in the emulator layer.

Our Solution: To enable KV API support in MoKE, we first analyze the chain of KV API requests, their transformation in the ADI layer, and the corresponding NVMe commands issued to the device, starting from the initialization of the device. We notice that the OpenMPDK KV library uses the three basic KV read, write, and delete commands to manage the device and key spaces instead of any admin or custom NVMe commands. The exact KV pairs that hold the key space list are pre-defined inside the KV API. We reverse engineer the implementation details of these KV pairs from the OpenMPDK KV API library. We create and insert these KV pairs in MoKE during initialization using the reverse-engineered keys and values. We also add support for multiple key spaces, along with key space isolation, in MoKE. As a result of these changes, applications can create, use or delete key spaces using the KV API layer. Next, we add support for KV I/O error handling by adding KV command error codes inside the emulator and transferring the error information from emulator to KV API using KV command context data structure. The above changes enable both regular and benchmark applications to interact with MoKE through the SNIA KV API.

B. Delay Emulation

Problem: The latency of an I/O operation in KV-SSD is an accumulation of two main components - request handling delay and data access delay. FEMU already implements a basic delay model for data access based on single-register and uniform page latency. However, it lacks delay emulation for variable-length KV data and KV request handling overheads (e.g. key hashing delay, K2P table processing delay, DMA transfer delay). To support variable-length KV data, we need to adapt the data backend and the FTL.

On the other hand, if we try to emulate the exact delays caused by KV request handling overheads, it would require frequent calls to the system clock, and may negatively influence delay emulation.

Our Solution: We modified FEMU's data backend and FTL to support variable-length KV data. Then, to tackle the problem of emulating KV request handling overheads, we consider a hybrid approach. We strive to emulate overheads incurred by different components in KV devices using different methods. First, we experiment with DMA transfer delays for large keys. Our experiments reveal that DMA transfer delays for command transfers are included in the latency inherently, because subsequent I/O commands are inserted into the NVMe submission queue at a slightly slower rate. Directly adding the DMA transfer delay to the I/O request submission time would result in counting the delay twice. Thus, we do not add any additional delay for the DMA transfer of large keys. Next, we measure the processing delay of different operations, such as key hashing and K2P table access. The average cumulative delay of these overheads for both read and write requests is around 4.2 and 4.4 microseconds, respectively, when the

99th percentile tail latency is ignored. When the tail latency is considered, the cumulative delay is about 4.8 and 16.2 microseconds for read and write requests. These measurements indicate that the delays added by different index operations are dependent on the type of requests (e.g., read or write or exist). Hence, we consolidate the delays incurred by different index operations and introduce four configurable variables T_{read}, T_{write}, T_{delete}, and T_{exist} to emulate the cumulative processing delays for read, write, delete and exist requests. These delays can be different for different index structures. Users can set these values through empirical studies on their own FTL implementation(s).

III. Accuracy Evaluation

We measure MoKE using the following configuration. We emulate a 64GB SSD (8 channels, 8 LUNs/channel, and 16KB flash page) where latency for program, read and erase operations are $660\mu s$, $45\mu s$, and $3.5ms$, respectively [5]. All experiments were performed on a machine with 2xIntel(R) Xeon(R) Silver 4208 CPU, 192GB DDR4 DRAM, with a 1TB SAS HDD as a backing store. OpenMPDK KVBench [14] is used to measure MoKE's performance (latency and throughput) because the microbenchmark tool is compatible with SNIA KV API and can generate realistic workloads [6], [13]. All experiments in this work use 10GB random read or write workloads (2.6M KV pairs with 16B keys and 4KB values), with queue depth of 1 for latency and 64 for throughput measurements similar to industry practice.

First, we observe the I/O latency with and without flash access delay emulation to verify if MoKE emulates delays (or no delays) correctly (Figure 1b and 1c). Average store and retrieve latency without delay emulation are $28.7\mu s$ and $29.5\mu s$, respectively. This delay is added partly by the host (KV driver, API, and KVBench itself) to process the request and partly by MoKE (as it cannot return immediately upon receiving the request even if the delay is set to 0). The unintentionally added processing delays by MoKE are small enough to be offset during actual I/O operations with (data placement and index) delay emulation. Second, we verify that MoKE's read latency can scale similar to Samsung KV-SSD, as we increase the queue depth [13]. MoKE can sustain read latency up to 16 queue depth, but suffers at very high queue depth due to the lack of large number of concurrent request managers, as seen in Samsung KV-SSD (Figure 1d). In summary, our experimental results show that MoKE is accurate enough to be used as a platform to trigger important emerging KV/object storage device-related research. We plan to continue enhancing MoKE to be more scalable and easy to use.

IV. Conclusions

MoKE is designed to fill the void of an inexpensive, extendable KV/object-SSD research platform. MoKE will likely foster thorough investigations related to the complex design choices within the algorithms performed and the data structures maintained inside the KV storage devices. Our evaluation results show that MoKE accurately mimics the performance trends of a real Samsung KV-SSD. We expect that MoKE will speed up future KV-SSD research and also enable researchers to explore other emerging directions such as namespace isolation, KV interface-based Computational Storage Devices (CSD), and many more.

Acknowledgment

This work was partially supported by National Science Foundation Awards CNS-2008324 and CNS-2008072 and KV-SSD equipment grant and Funded Research Collaboration with Samsung MSL GRO (Global Research Outreach).

References

[1] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.

[2] Tai Chang, Jen-Wei Hsieh, Tai-Chieh Chang, and Liang-Wei Lai. Emt: Elegantly measured tanner for key-value store on ssd. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(1):91–103, 2022.

[3] NVM Express. NVMe Specification 2.0, 2022. https://nvmexpress.org/developers/nvme-command-set-specifications/.

[4] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. Kaml: A flexible, high-performance key-value ssd. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–384. IEEE, 2017.

[5] Dongku Kang, Woopyo Jeong, Chulbum Kim, Doo-Hyun Kim, Yong Sung Cho, Kyung-Tae Kang, Jinho Ryu, Kyung-Min Kang, Sungyeon Lee, Wandong Kim, Hanjun Lee, Jaedoeg Yu, Nayoung Choi, Dong-Su Jang, Jeong-Don Ihm, Doogon Kim, Young-Sun Min, Moo-Sung Kim, An-Soo Park, Jae-Ick Son, In-Mo Kim, Pansuk Kwak, Bong-Kil Jung, Doo-Sub Lee, Hyunggon Kim, Hyang-Ja Yang, Dae-Seok Byeon, Ki-Tae Park, Kye-Hyun Kyung, and Jeong-Hyuk Choi. 7.1 256gb 3b/cell v-nand flash memory with 48 stacked wl layers. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 130–131, 2016.

[6] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. Towards building a high-performance, scale-in key-value storage system. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, page 144–154, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Sang-Hoon Kim, Jinhong Kim, Kisik Jeong, and Jin-Soo Kim. Transaction support using compound commands in key-value SSDs. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2019.

[8] KOSP. Kinetic Open Storage Project, 2015. https://github.com/Kinetic.

[9] Changgyu Lee, Hyeongu Kang, Donggyu Park, Sungyong Park, Kim Youngjae, Jungki Noh, Woosuk Chung, and Kyoung Park. ilsm-ssd: An intelligent lsm-tree based key-value ssd for data analytics. pages 384–395, 10 2019.

[10] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S Gunawi. The {CASE} of {FEMU}: Cheap, accurate, scalable and extensible flash emulator. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 83–90, 2018.

[11] Pratik Mishra, Rekha Pitchumani, and Yang Suk Kee. Learnings from an under the hood analysis of an object storage node io stack. 2022.

[12] Dan Robinson (The Resgister). SK hynix and Los Alamos Labs to demo key-value store accelerating SSD, 2022. https://www.theregister.com/2022/08/01/sk_hynix_lanl_kv_csd/.

[13] Manoj P. Saha, Adnan Maruf, Bryan S. Kim, and Janki Bhimani. Kv-ssd: What is it good for? In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1105–1110, 2021.

[14] Samsung. OpenMPDK KVSSD, 2018. https://github.com/OpenMPDK/KVSSD/.

[15] SNIA. Key value storage API specification, 2020. https://www.snia.org/keyvalue.