2023 31st International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS) | 979-8-3503-1948-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/MASCOTS59514.2023.10387663

OctoKV: An Agile Network-Based Key-Value Storage System with Robust Load Orchestration

Yeohyeon Park[1], Junhyeok Park[1], Awais Khan[2], Junghwan Park[1], Chang-Gyu Lee[1]
Woosuk Chung[3], Youngjae Kim[1*]
[1]Sogang University, Seoul, Republic of Korea, [2]Oak Ridge National Laboratory, [3]SK hynix

***Abstract*—In this paper, we propose OctoKV, an innovative network-based key-value storage system. OctoKV addresses the repetitive address translation overhead associated with traditional key-value stores running on file systems on the client side. To mitigate this overhead, we implemented the key-value store on the server side using NVMe-oF and a user-level NVMe driver. In particular, we employed fine-grained resource monitoring and load balancing based on heuristics to optimize I/O performance. OctoKV is deployed on a Linux cluster with Intel SPDK. The extensive evaluation shows that OctoKV achieves lower I/O response times in comparison to traditional approaches where key-value stores run on the client side. Also, the proposed load balancing strategies efficiently enhance I/O response times by equally distributing the workload from overloaded cores to other cores.**

***Index Terms*—High Performance I/O, Key-Value Store, Storage System, I/O Scheduling**

I. Introduction

Disaggregated storage is becoming increasingly popular in modern computing environments, necessitating the adoption of Key-Value Stores (KVS) due to its proven superiority of key-based lookup, high performance, and scalability. While research on network in-memory KVS has been proposed in cluster environments [1, 2, 3], these solutions have primarily focused on designing in-memory caches for fast read operations, rather than exploring persistent storage capabilities.

The adoption of KVS in network-based persistent storage systems, such as Storage Area Networks (SANs), has been relatively neglected. The traditional KVS, such as RocksDB and LevelDB, running on a file system in a SAN environment introduce various overheads at multiple layers. For instance, when a client sends a KV request, file system operations incur overheads due to the conversion of key-value pairs to files and files to block addresses. Further, running KVS atop the file system introduces OS overheads like user-kernel mode switching, interrupt handling, and context switching.

To overcome such multiple address conversion and OS-related overhead, Key-Value SSDs [4, 5, 6, 7] have implemented the KV storage engine directly within the SSDs. This design allows applications to directly access the SSD, bypassing the cumbersome OS stack. An alternative in parallel, the Intel Storage Performance Development Kit (SPDK) [8] has revolutionized storage system design, providing a framework specifically tailored to harness the full potential of ultra-low latency NVMe SSDs. SPDK user-level NVMe driver with polled mode significantly mitigates the performance overheads associated with traditional OS. By reducing context switching between user and kernel modes and eliminating expensive interrupts, SPDK unlocks the true capabilities of NVMe SSDs, pushing the boundaries of storage performance. Due to the superiority of SPDK, efforts have been made to adopt SPDK to build high-performance KVS for single node systems such as SpanDB [9], EvFS [10] and TridentKV [11]. Further, SPDK extends support to NVMe over Fabrics (NVMe-oF), enabling the development of disaggregated storage architectures that offer exceptional performance and scalability.

Based on these foundations, there exists an intriguing opportunity to leverage SPDK capabilities in constructing a network-based KVS. Therefore, it seems trivial to mitigate OS and file system-related overheads by extending a network-based KVS using SPDK. However, with careful investigation of the shared-nothing design employed by SPDK, we uncover critical load-imbalance problems and reveal lack of a data structure sharing between cores, resulting in load disparities, particularly in skewed workloads. Specifically, we identify two pivotal challenges as follows;

First, there is a load imbalance issue among TCP connections, where clients send NVMe commands over TCP to the server. While clients establish multiple TCP connections, and the server runs corresponding SPDK threads to process these commands, the workload distribution among connections is uneven. This imbalance results in some connections receiving a heavier workload, leading to delays in I/O response time.

Second, running the KV storage engine on the server increases the CPU load, and oftentimes contributes to the load imbalance problem. Notably, hash and LSM-tree-based KV storage engines impose CPU-intensive tasks such as executing hash computations and merge-sorts during compaction. Such added CPU burden further contributes to delays in response time, with the TCP layer unaware of these application processes. Therefore, threads on connections experiencing high load significantly impact the overall response time.

Therefore, in this work, we introduce OctoKV, an agile server-side key-value store, leveraging the robust capabilities of Intel SPDK for disaggregated storage architectures. Notably, OctoKV avoids the kernel overhead associated with operating on top of the OS, as well as eliminates the file system overhead related to multiple address conversions. OctoKV addresses the load imbalance problems by designing an I/O event migration scheme to distribute the load of I/O between SPDK cores. The key idea is load-aware fine-grained scheduling. To the best of our knowledge, our work is the first to migrate events between cores for CPU load balancing in SPDK.

We conduct experiments on two servers connected via

*Y. Kim is the corresponding author.

a 10Gbps Ethernet, using a slightly modified version of RocksDB db_bench benchmark to evaluate the performance of OctoKV. Based on results, OctoKV shows 12% lower I/O response time in "Fill Random" workload compared to in-house hash-based KVS with EXT4 file system mounted on block-based NVMe-oF device. Further, the proposed IO event migration scheme shows 12% reduction in I/O response time compared to a baseline OctoKV without migration scheme.

II. Background

A. Event-Driven User-level NVMe Drivers

Intel SPDK [8] is one of the most popular storage frameworks for building a high-performance storage system with ultra-low latency NVMe SSDs. To achieve low latency and high throughput, SPDK offers a user-mode NVMe driver with polled-mode. The NVMe driver enables SPDK to mitigate additional latency in the I/O response time of fast SSDs. This latency arises when the SSD latency is lower than the time needed for interrupt and getting completion. Moreover, SPDK employs a sophisticated per-core lock-less event framework to effectively alleviate the potential lock contention arising from shared memory among cores, thus achieving optimal performance. Consequently, the client I/O request is intricately bound to a single core, with the request seamlessly divided into multiple events that are meticulously processed in a sequential manner by the designated core.

In SPDK, BDEV serves as a fundamental component of the SPDK block device layer. Each event within the system must be processed by the BDEV. BDEV offers a pluggable module API for implementing block devices that interface with block storage devices. Users have an option to utilize existing BDEV modules or create virtual BDEV (VBDEV) modules, which enable the construction of block devices on top of an existing BDEV. These BDEV operations collectively constitute a comprehensive I/O request. Through the virtual BDEV mechanism, SPDK can generate events for other BDEVs. Users have the freedom to modify VBDEV as a user-defined function and insert it within the I/O path to perform necessary operations such as compression or encryption during I/O processing. Each core operates a dedicated thread known as a Reactor, which sequentially processes multiple pollers (functions).

B. Network-Based Block Storage

NVMe-oF is a protocol designed to connect hosts to storage across a network fabric using the NVMe protocol. It enables data transfers between a host and target SSD or system over a network through an NVMe command. The data corresponding to the NVMe command is transferred through networks such as Ethernet, Fibre Channel (FC), or InfiniBand. The users connect the remote NVMe SSD through the NVMe-oF as a local NVMe SSD, allowing fundamental storage disaggregation. Note that, this work is targeted for such storage disaggregation environments. SPDK supports the NVMe-oF for both RDMA and TCP transports with providing NVMe-oF hosts for client-side NVMe drivers and targets for server-side. A typical KVS runs on the file system of block storage (local or network block storage). On the server, SPDK runs the NVMe-oF target BDEV, and NVMe driver BDEV to handle the I/O request from the client. The NVMe-oF target BDEV receives the client's NVMe commands through socket queues. The NVMe driver BDEV generates block I/O and delivers it to the SSD.

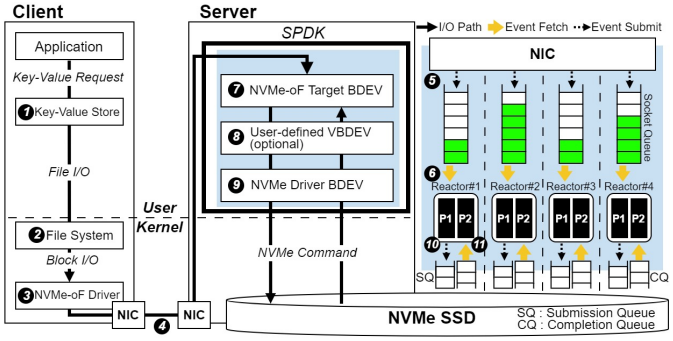
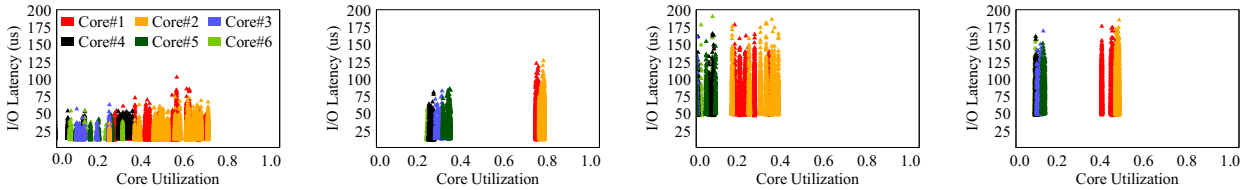


Fig. 1. An illustration of IO flow in network-based block storage using SPDK.

Figure 1 depicts the software stack when running a KVS on SPDK-based network block storage and demonstrates that how a server-side SPDK processes I/O requests from the client. When the server is running, SPDK spawns N SPDK threads, also referred to as Reactors, with the value of N determined by the user. Each thread (Reactor) is assigned to a dedicated core in a 1:1 mapping. The Reactor, bound to the core of SPDK, uses two pollers (functions) in a round-robin order to poll the queues required for I/O processing. Specifically, the Reactor sequentially executes Poller1 (P1) to poll the socket queue containing NVMe commands and Poller2 (P2) to poll the completion queue holding completed I/Os from the SSD.

Next, we describe how Reactor utilizes pollers to handle the client's I/O requests. ❶ The application sends a Get/Put KV request to the key-value store. The key-value store converts the key-value request into file I/O request and forward it to the kernel-space file system. ❷ The file system converts file I/O request into block I/O requests and passes them to the NVMe-oF driver. ❸ The NVMe-oF driver converts block I/O into NVMe commands and ❹ forwards to the NVMe-oF target using the NVMe-oF protocol. ❺ The NVMe command sent by the client is stored in the socket queue of the NVMe-oF target. ❻ Subsequently, the SPDK Reactor sequentially executes the pollers. Firstly, the poller (P1) checks the socket queue, dequeuing an item (the NVMe command sent from the client) if it is not empty. ❼ Secondly, P1 sequentially processes the NVMe command by executing the NVMe-oF target BDEV, any user-defined (V) BDEVs, and the NVMe driver BDEV. The NVMe-oF target BDEV converts the NVMe command received from the client into the SPDK's I/O format (bdev-io). ❽ Users can perform desired processing on I/O through user-defined VBDEV. ❾ Then, the NVMe driver BDEV converts the bdev_io into block I/O and ❿ enqueues it into the submission queue. ⓫ The Reactor proceeds to execute the next poller, P2. P2 examines the completion queue of the NVMe driver to determine if there are any completed I/O operations from the SSD. If there are completed operations present in the queue, P2 dequeues and processes them accordingly. In the event that the completion queue is empty, the Reactor resumes executing P1 (❻). Else, P2 notifies the NVMe driver BDEV and the NVMe-oF target BDEV through callbacks that the I/O operations are completed.



(a) Light Workload (Put) (b) Heavy Workload (Put) (c) Light Workload (Get) (d) Heavy Workload (Get)

Fig. 2. The analysis of the correlation between core utilization and I/O latency.

III. Motivation

In OctoKV, the client requests (NVMe commands) are sent through NVMe-oF to the server. The server utilizes SPDK user-level NVMe driver, which employs a per-core lock-less event framework. This framework enables fast I/O processing without the need to share data structures or connections among SPDK threads. On the server, NVMe-oF target establishes TCP connections with the client, with a number of connections equal to the number of Reactors.

Note that, we highlighted the concerns regarding load balancing in the SPDK NVMe-oF framework in Section I. Here, we conducted experiments to empirically verify these issues and their impact.

1) Queue Depth Analysis: We conducted a series of experiments to verify that the SPDK's NVMe-oF framework exhibits a lack of load balancing across server cores during I/O request processing. The experiments were performed on two servers connected by a 10Gbps network, where we used light and heavy workloads on the clients for both Put and Get requests, respectively. These requests were forwarded to the server using NVMe-oF. Further details on experimental setup can be found in Section V-A. We measured the queue depth, which represents the number of outstanding NVMe commands, on each core/SPDK thread TCP connection (socket queue) for each workload execution. Note that, the experimental results in Table I do not include the key-value storage engine overhead.

We observed that increasing the workload for Put requests results in a more significant load imbalance. For light workload (Put), the standard deviation was 0.78. However, under heavy workloads, the deviation increased by approximately 2.2 times to 1.70 as shown in Table I. Specifically, in heavy workload, core#1 and core#2 processed about 2 to 3 times more I/O requests compared to the other cores. Notably, the queue depth is larger for Get compared to Put workload. We noticed a similar trend as in Put, i.e., load imbalance becomes more severe for heavier Get workload.

2) Core Utilization vs I/O Latency: The queue depth has a positive correlation with the CPU utilization of the respective core. We measured the correlation between I/O latency and CPU utilization and report the results in Figure 2. Note that, the I/O latency is measured by the server, rather than the I/O response time from the client. Overall, it shows an increasing trend of I/O latency as core utilization increases. Figure 2(a)&(b) show the results for Put workloads. As observed in Table I, in light workload (Put), core utilization is evenly distributed from 0.0 to 0.8. On the contrary, heavy workload (Put) show a significant difference in queue depth between cores, resulting in a bimodal distribution of core utilization between cores. As a result, high core utilization means that SPDK threads must handle a larger number of I/Os,

TABLE I

Average Queue depth (QD) with NVMe commands per core. C(i) represents the core with core ID=i. LW and HW denote light and heavy workload, respectively.

QD	C(1)	C(2)	C(3)	C(4)	C(5)	C(6)	Avg	Stdev
LW(Put)	2.00	2.21	0.75	1.58	0.67	0.33	1.26	0.78
HW(Put)	5.25	5.48	2.00	2.06	2.13	2.13	3.18	1.70
LW(Get)	3.95	4.23	1.27	1.36	2.00	1.82	2.43	1.31
HW(Get)	6.06	6.54	2.69	2.62	2.92	2.65	3.91	1.86

TABLE II

CPU load based on the execution of the KV storage engine.

	Light Workload (LW)		Heavy Workload (HW)	
	Baseline	OctoKV	Baseline	OctoKV
Avg	0.315	0.339 (7.7%)	0.436	0.491 (12.6%)
Stdev	0.236	0.263 (11.4%)	0.230	0.213 (-7.3%)

consuming more CPU cycles, thus increasing I/O latency. For example, the average I/O latency of the core with the highest core utilization, core #1, is 38µs, while the average I/O latency of the core with the lowest core utilization, core #6, is 19µs. In other words, core #1 exhibits approximately twice the I/O latency compared to core #6. Thus, there is a high potential for balancing the arrival of I/O requests between cores with high core utilization and cores with low core utilization, which can lead to improved response times.

Figure 2(c)&(d) show the results for Get workload. For the light workload (Get), the core utilization is evenly distributed from 0.0 to 0.4. Whereas, the heavy workload (Get) exhibits a bimodal distribution. Similar observations were obtained as those shown in Figure 2(a) and (b).

3) Increase in CPU Load: To further study the increase in CPU load and load imbalance, when running the KV storage engine on the server, we measured the average and standard deviation of CPU utilization in two scenarios: i) running the KV storage engine (OctoKV) and ii) not running it (baseline). We report the results in Table II. The percentage (%) in results indicates the growth rate. For the light workload, OctoKV shows an approximate 7.7% increase in CPU load compared to the baseline. Additionally, it demonstrates an increase of approximately 11.4% in the variance of CPU core utilization. On the other hand, for the heavy workload, as expected, OctoKV exhibits an increase of approximately 12.6% in CPU load compared to the baseline. However, since the heavy workload already has high utilization of cores, it actually shows a decrease of approximately 7.3% in the variance of CPU core utilization. Nevertheless, there is still an imbalance in CPU core utilization.

IV. OctoKV: Proposed System

A. System Overview

OctoKV is a network-based server-side KV storage system, with multiple clients allowed to access the storage through it.

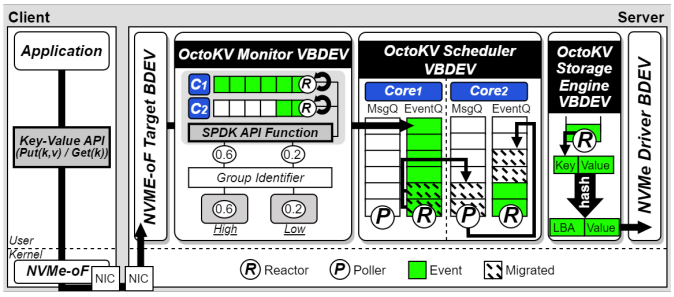


Fig. 3. An overview of OctoKV.

A client can host multiple applications, with each application using the key-value APIs to access the storage. Furthermore, in OctoKV, the application has the capability to bypass the kernel. And, the server runs SPDK which includes OctoKV Storage Engine (OKSENG), OctoKV Monitoring (OKM), and OctoKV Scheduler (OKS) modules to enable a robust and efficient I/O load orchestration. Figure 3 presents the overview of OctoKV. The modules between the NVMe-oF target BDEV and the NVMe driver BDEV are server-side components of OctoKV, and implemented as VBDEVs in SPDK.

The OKSENG manages key-value pairs to enable clients to read and write on the NVMe SSD in the server, and it adopts data structures typically employed by KVSs such as hash or LSM-tree. The OKM monitors the utilization of each core at a fixed time interval. Note that, the interval window for collecting per core utilization is a tunable parameter and can be tuned as per desired configurations. Afterward, the OKM analyzes the per-core utilization to determine if any cores are overloaded. If the OKM identifies the load imbalance problem, it informs the OKS to make robust scheduling decisions to efficiently distribute the load among the cores. Specifically, without making shared space between cores, OKS uses a messaging framework, such as messages and message queues to migrate I/O operations/events from busy cores to idle cores (refer to MsgQ in Figure 3).

B. Client API and Driver Module

We developed a key-value API library, i.e., Put() and Get(), which allows user-level applications to interact with network-based key-value storage using the NVMe protocol. The user requests NVMe I/O commands directly to the NVMe Driver through NVMe I/O passthrough. Moreover, to send key-value operations (Put and Get) through the NVMe command, we extended the existing NVMe protocol. Specifically, we store the operation identifier (Put or Get) in the opcode area of the NVMe command and save the key in the LBA address area. Note that, we specified the buffer size, which represents the size of the buffer used to store data to be sent (Put) or data to be received (Get), in the reserved area of the NVMe command.

C. OctoKV Storage Engine

We implemented OctoKV Storage Engine (OKSENG) in SPDK to minimize I/O software overhead on the server, i.e., bypassing the storage server's kernel stack. The clients access key-value pairs on the network as if they are in local KVS. Note that, for the OKSENG, existing data structures such as hash and LSM-tree can be employed. We implemented a hash-based KVS to validate the usefulness of our proposed server-side key-value store approach and avoid some unpredictable performance degradation caused by compactions of the LSM tree.

The OKSENG comprises three primary modules: (i) a key-based hash function (SHA-1), (ii) a bitmap array for efficient management of empty space on NVMe SSDs, and (iii) a hash table that maintains logical block addresses and their corresponding key-value pairs on the SSDs. To ensure thread-safety, mutex locks are employed to protect shared data structures. However, locking the entire array or table with a single lock results in lock contention and limited concurrency. To mitigate this performance issue, we partition the bitmap array and hash table into 2^n units, where n is the number of bits used for partition selection, and perform critical section protection for each partition. The partition to use for a given key is determined by its hash value. The n most significant bits of the hash value are used to select the starting partition for the bitmap array and hash table. The next 2^n bits of the hash value are used to determine the index in the hash table. The size of the hash value we use is 20B.

Meanwhile, the partition of the bitmap array tracks the availability of empty blocks in the partition itself, after dividing the total capacity of the storage device by 2^n. Using the first-fit algorithm, if the required number of consecutive empty blocks are found in the partition, the logical block address of the starting block is assigned. We set n to 4 for all the experiments conducted in Section V.

D. OctoKV Monitoring Module

The OKM VBDEV measures each core utilization in a fixed time interval window (W). Fortunately, SPDK provides a set of functions to analyze and report the CPU cycles, performing real tasks (excluding pollings). Therefore, OKM uses these functions to obtain an array of consumed CPU cycles for each core, and then performs internal computations for scheduling. And ultimately, OKS performs load balancing among the cores if the computation satisfies the following two conditions at each time interval window (W).

- **CPU Overloading:** This condition determines whether there is at least one overloaded core in the system. OKM compares the utilization of each core with a user-defined threshold value (T_{OL}), which ranges from 0.0 to 1.0. Setting T_{OL} too low may incorrectly judge a core to be heavily loaded even if it is not, while setting it too high may incorrectly judge a heavily loaded core to be underutilized. Therefore, it is crucial to properly configure T_{OL} based on the specific system environment.
- **Load Imbalance:** After satisfying the overloading condition mentioned above, OKM proceeds to check for load imbalance among the cores using the following equation: $Max\{(f_{cutil}(C))\} - Min\{f_{cutil}(C)\} > T_{LB}$. In this equation, $f_{cutil}(c_i)$ represents the utilization of core c_i, where i ranges from 1 to n. The core group C consists of n cores (c_1 to c_n). The equation calculates the difference between the maximum and minimum core utilization values of all cores and compares it with a predefined threshold

value (T_{LB}), which ranges from 0.0 to 1.0. Similar to T_{OL}, T_{LB} is set by the user and, if not properly configured, may lead to incorrect judgments of load imbalance occurrence. Therefore, both T_{OL} and T_{LB} must be properly set.

When both of these conditions are satisfied, OKM categorizes the cores into two groups based on their utilization: a group with high core utilization and a group with low core utilization. We use the average utilization of all cores (U_{avg}) as the criterion to differentiate between these two groups. During load balancing, the OKS migrates I/O requests from cores in the high group to cores in the low group. In the previous description, we referred to two groups as high and low. However, our approach outlined above can be easily extended to accommodate more than two groups.

Meanwhile, the OKM operates on a single core at a time, dynamically selecting the core with the lowest utilization for each time interval (W). The initial core selected is core #1 of the system. Only the Reactor with the lowest utilization actually executes the OKM. And, the per-core utilization data and group configuration provided by the OKM can be globally accessed by every OKS VBDEV of each Reactor.

E. Load-Aware Balanced I/O Scheduling

When the cores are not balanced correctly, I/O requests processed on overloaded cores may experience delays due to long wait times in the event queue. In order to mitigate this issue, OKS dynamically migrates I/O requests from overloaded cores to cores that are not overloaded at runtime. This proactive load-balancing strategy reduces wait times and enhances overall system performance. If both the CPU overloading condition and the load imbalance condition are met, OKM triggers the OKS to perform migrations for load balancing.

When processing an I/O request from the client on the server, OctoKV goes through the following three stages: NVMe-oF target, KV store, and NVMe driver. The NVMe-oF target retrieves an I/O request in the NVMe-oF format that arrived on the TCP connection and converts the request into multiple I/O (NVMe) commands to enable the user-level NVMe driver to process it. The KV store (OKSENG VBDEV) involves indexing-related operations. As described in Section IV-C, the hash function is executed for the key, and the put/get operation is performed on the index data structure. Finally, the NVMe driver sends NVMe commands directly to the SSD and waits for a callback. All the operations executed in the KV store and NVMe driver are managed as I/O events of SPDK.

However, the NVMe-oF target stage must execute on the core where the I/O request arrives. It is because NVMe commands generated from the received operation need to be converted into SPDK events, enabling them to be properly handled within the SPDK event handling process. The SPDK event handling process includes pushing events to the message queue for migration, which will be discussed in detail shortly. Thus, the NVMe-oF target stage is initially executed on the same core it arrives. On the other hand, the remaining stages do not have this limitation since they are already SPDK events. Thus, they can be freely moved to other cores for execution at anytime.

The CPU cycle consumption of each stage varies depending on the system configuration and the I/O access pattern. If the system is supported by fast networks, such as 100Gbps Ethernet, the NVMe-oF target stage could take a shorter amount of time compared to the other stages. If the embedded KV store is based on the LSM-tree structure and compactions occur, then the KV store stage could be longer than others. Therefore, due to the variable nature of this consumption, we introduce notations for the consumption of each stage: R_{oF}, R_{KV}, R_D, average percentage of how much each stage accounts for the total time, respectively. Note that, the NVMe-oF target stage cannot be moved, whereas other stages can move. Therefore, the maximum benefit obtained from migrating events is a ($R_{KV} + R_D$) decrease in the migrating core utilization.

1) How many I/O events for migration?: Next, we discuss the basis for the algorithm that utilizes thresholds to determine the number of events that can be migrated from high-utilization group cores to low-utilization group cores. This decision is crucial as transferring an excessive number of I/O events can potentially overload the cores in the low-utilization group.

For simplicity, we assume that (i) each group has only one core, and (ii) the core (C_{high}) in the high group has a core utilization of U_{high} and total N_{high} events in the KV store and NVMe driver stages, while the core (C_{low}) in the low group has a core utilization of U_{low} and total N_{low} events in the KV store and NVMe driver stages.

Our specific interest is to find out how many events among N_{high} events should be migrated from C_{high} to C_{low}. Note that even if all N_{high} events in the KV store and NVMe driver stages of C_{high} are moved to C_{low}, the decrease in U_{high} is at most ($R_{KV} + R_D$) of U_{high} since R_{oF} of total time cannot be reduced due to the aforementioned limitation. If U_{low} is low enough so that U_{low} + U_{high} $\times$ ($R_{KV} + R_D$) is still below a certain threshold (T), we move all N_{high} events from C_{high} to C_{low}. However, if U_{low} + U_{high} $\times$ ($R_{KV} + R_D$) is above the threshold (T), we calculate the following ratio (R): $\frac{T - U_{low}}{U_{high}(R_{KV}+R_D)}$, ($0.0 \leq R \leq 1.0$). Then, we migrate N_{high} $\times$ R events from C_{high} to C_{low}. Here, we use the threshold as U_{avg}. In our experimental environment which supports the 10Gbps Ethernet and implements the simple hash-based KV store, the NVMe-oF target stage consumes approximately 80% (R_{oF}) of core utilization, while the remaining 20% (R_{KV} + R_D) is taken up by the KV store and NVMe driver stages. Until now, we assumed that each group has only one core, but even when each group is composed of multiple cores, the proposed algorithm can be easily extended based on it.

2) Scheduling Heuristics: Here, we determine which core in the low group the events need to be migrated to. The key objectives of OKS are: (i) to minimize the I/O response time, and (ii) to reduce the overhead of making decisions for scheduling. We pointed out that minimizing the time spent for making scheduling decisions is critical as spending too much time on the decision making causes to increase the I/O response time even after load-balancing. Therefore, we propose two scheduling strategies that attempt to strike a balance between these two competing objectives based on heuristics.

- *RoundRobin (RR):* Tasks are assigned to each core in the

TABLE III
HARDWARE/SOFTWARE SPECIFICATIONS OF SERVERS.

CPU	Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz
	(Client) 10 cores @ 2.40GHz, (Storage) 6 cores @ 1.80GHz
Memory, Disk	32GB DDR4, 500 GB Samsung 970 EVO SSD
Interface	NVMe-oF (10 Gbps Ethernet)
Software	Ubuntu 20.04, SPDK v.21.10, RocksDB v.6.23

low group in a round-robin order, regardless of the utilization of each core. For example, when the $f_{cutil}()$ values of core#3 and core#4 are 0.2 and 0.3, the RR algorithm assigns tasks to both cores in a 1:1 ratio.

- *Proportional Share (PS):* Tasks are placed in proportion to how much each core in the low group can handle without being overloaded. Suppose, the core utilization $f_{cutil}()$ available for each core in the low group is U_{avg} - $f_{cutil}()$. For example, when $f_{cutil}()$ values of core#3 and core#4 are 0.2 and 0.3 and U_{avg}=0.4, the PS algorithm assigns tasks to core#3 and core#4 in a ratio of 2:1.

F. Implementation

We implement the migration in message passing infrastructure of SPDK [12]. SPDK adopts a message passing method instead of a traditional locking method to enable concurrency in multi-threaded programming and to achieve linear scalability with the addition of storage devices. In SPDK, each Reactor thread has its own message queue, allowing multiple Reactors to communicate with each other by sending events as messages to each other's message queues. As part of this mechanism, the Reactor checks its message queue for any pending messages to retrieve between the executions of two pollers, P1 and P2 (refer to Section II-B). Therefore, we employ it to implement the proposed load balancing algorithm. We have made slight modifications to the message queue poller routine of each Reactor to execute OKSENG VBDEV for migrated events, thus enabling the utilization of per-core message queues as channels for event migrations. Subsequently, the OKS of Reactor in the high utilization group pushes events to the message queue of the target low utilization cores, considering the calculated number of event migrations. Figure 3 demonstrates this mechanism clearly.

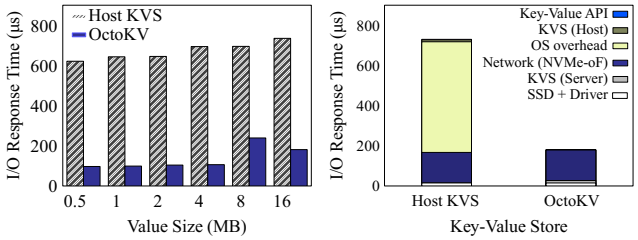
V. EVALUATION

A. Evaluation Setup

We implemented OctoKV[1] using SPDK v.21.10, with two servers connected via 10 Gbps Ethernet. The client communicates with the storage server through NVMe-oF. Both the client and storage servers have the same specifications, but they differ in terms of CPU count and clock speed (refer to Table III). In general, the CPU capability of storage servers tends to be lower compared to that of hosts.

For the performance evaluation, we slightly modify the RocksDB db_bench benchmark to directly send NVMe command requests to OctoKV. The db_bench is a multi-threaded benchmark, which allows multiple threads to simultaneously insert or retrieve key-value pair. Each thread executes 300K put or get operations synchronously.

[1]https://github.com/lass-lab/octokv



(a) I/O Response Time (b) Time Breakdown

Fig. 4. Comparing the measurement of I/O latency performance for Put workloads and the I/O response time breakdown for a value size of 16KB.

We defined three workloads based on the number of threads: Light workload (7 threads), Medium workload (10 threads), and Heavy workload (12 threads). We used "Fill Random" and "Read Random" options for Put (write) and Get (read) workloads, respectively. We used a key size of 4B and a value size of 16KB as defaults. We measured I/O response time, I/O latency time, throughput, and the utilization of individual CPU. Note that the I/O latency is measured by the server, whereas the I/O response time is measured from the client's perspective. Unless stated otherwise, the average mentioned refers to the average of three runs. We compared the following three systems:

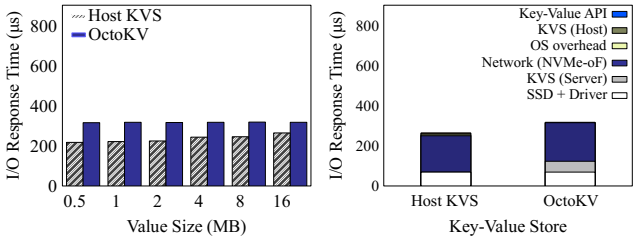
- **Host KVS:** In order to assess the I/O performance impact of OctoKV in a fair manner, we developed a host-side key-value store that incorporates the same hash-based key-value storage engine used in OctoKV. A detailed description of the Host KVS is provided in the following section.
- **OctoKV** or **OctoKV-LB:** OctoKV is the proposed system with only KV storage engine running on the storage server. In contrast, OctoKV-LB is OctoKV with the load-aware balanced I/O scheduling.

OKM collects the utilization of all cores every W, and W is set to 1 second. It takes 180µs to collect once which is negligible in terms of overhead.

B. Performance Evaluation

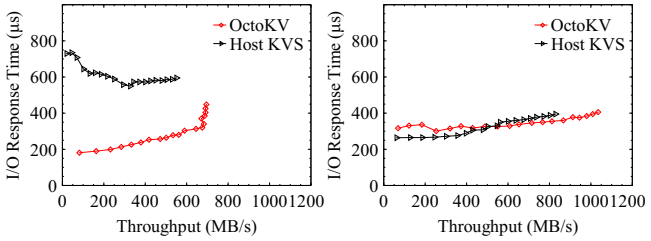
Figure 4 shows a comparison of the I/O response time between Host KVS and OctoKV for the Put workload. For a fair comparison on par with OctoKV, Host KVS invokes the fsync() function each time it writes a key-value pair to the DB file, ensuring durability. Furthermore, to minimize the overhead caused by sharing DB files among I/O threads in particular when fsync() is called, the Host KVS creates multiple DB files, thereby ensuring minimal sharing of the same DB file among the IO threads. Figure 4(a) shows the response times according to the value size. As expected, the I/O response time of the Host KVS is approximately 3 to 6.4 times higher than that of OctoKV. To analyze the difference in I/O response times in more detail, we compared the time breakdown for a value size of 16KB. From the result in Figure 4(b), we found that the OS overhead accounts for approximately 75% of the Host KVS's I/O response time. On the other hand, OctoKV completely eliminates such overhead, leading to a significant reduction in I/O response time.

Figure 5 presents the results for the Get workload. In contrast to the results for the Put workload, OctoKV exhibited



(a) I/O Response Time (b) Time Breakdown

Fig. 5. Comparing the measurement of I/O latency performance for Get workloads and the I/O response time breakdown for a value size of 16KB.



(a) Put workload (b) Get workload

Fig. 6. Comparing the I/O response time and throughput of the Host KVS and OctoKV for various I/O loads. A value size of 16KB was used.

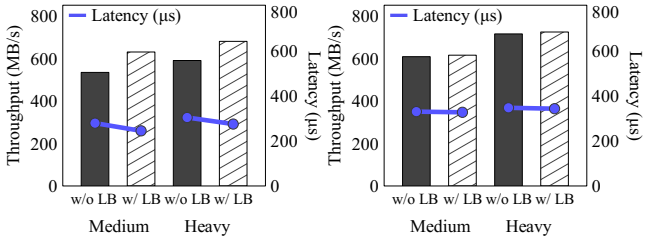
approximately 1.2 to 1.4% higher response time on average than the Host KVS, regardless of the value size (refer to Figure 4(a)). The OS overhead in the Put workload contributes significantly compared to the Get workload but network communication overhead shows over 50% (refer to Figure 5(b)).

Additionally, OctoKV takes approximately 2.5 times longer to execute the key-value storage engine compared to the Host KVS (refer to KVS). This is because OctoKV has a lower CPU clock speed than the KVS (refer to Table III). Due to this reason, the overall I/O response time of the Host KVS is lower than that of OctoKV. However, this experiment focuses on measuring the performance of directly reading data from disk without any caching effects on the host. Note that, systems with host-side caching will show little performance difference.

Next, we measured the throughput and I/O response time of Host KVS and OctoKV by increasing I/O loads from the client. Figure 6(a) shows the results for the Put workload. As expected, the Host KVS exhibits higher I/O response time compared to OctoKV, and a lower maximum throughput than OctoKV. This suggests that under heavy I/O load, the Host KVS struggles to process incoming I/O requests, potentially leading to overflow. On the other hand, OctoKV appears to be more capable of handling such workloads. In Figure 6(b), the results for the Get workload are presented. Both the Host KVS and OctoKV demonstrate similar I/O response times as the I/O load increases. However, the Host KVS exhibits a lower throughput of approximately 200MB/s compared to OctoKV.

C. Load-aware Balanced I/O Scheduling

1) Effectiveness of Load-balancing Algorithm: Figure 7 provides a detailed analysis of the performance benefits of load-balancing I/O scheduling using the PS heuristic scheduling for medium and heavy workloads. Figure 7(a) shows the performance for the "Fill Random" workload. Figure 7(a) medium workload result shows that OctoKV-LB outperforms OctoKV by 18% in terms of throughput.



(a) Put workload (b) Get workload

Fig. 7. Comparative analysis of throughput and I/O latency for medium and heavy workloads with "Fill Random" and "Read Random" options. OctoKV-LB used T_{OL}=0.4, T_{LB}=0.1, interval(W)=1, and a PS algorithm.

TABLE IV
PER-CORE EVENT COUNTS OF OCTOKV FOR (HEAVY, PUT).

Event	C(1)	C(2)	C(3)	C(4)	C(5)	C(6)	Sum
NVMe-oF Target	26511	30412	10441	11844	11035	9757	100000
KV Store	26511	30412	10441	11844	11035	9757	100000
NVMe Driver	26511	30412	10441	11844	11035	9757	100000
Sum	31323	91236	79533	35532	33105	29271	300000

TABLE V
PER-CORE EVENT COUNTS OF OCTOKV-LB FOR (HEAVY, PUT).

event	C(1)	C(2)	C(3)	C(4)	C(5)	C(6)	Sum
NVMe-oF Target	26394	30502	10394	11575	11583	9552	100000
KV Store	0	0	25024	25323	25231	24422	100000
NVMe Driver	0	0	25024	25323	25231	24422	100000
Sum	26394	30502	60442	62221	62045	58396	300000

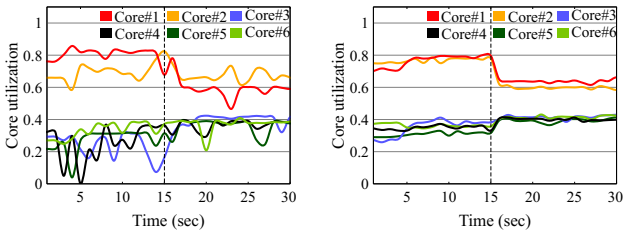
Furthermore, OctoKV-LB exhibits 12% lower I/O latency than OctoKV. Similarly, in Figure 7(a) heavy workload, OctoKV-LB demonstrates 15% higher throughput than OctoKV. Additionally, OctoKV-LB shows 10% lower latency than OctoKV. These results are consistent with those shown in Figure 6(a) median workload, demonstrating the significant performance gain achieved through load balancing.

Figure 7(b) shows the performance results for the "Read Random" under the same conditions as the previous experiment. As shown in Figure 7(b) medium workload, OctoKV-LB exhibits 1% higher throughput than OctoKV. The latency for OctoKV-LB is 1% lower than OctoKV. Similarly, in Figure 7(b), OctoKV-LB shows 1% higher throughput than OctoKV in heavy workload. OctoKV-LB shows 1% lower latency than OctoKV. Overall, the evaluation confirms that the load-aware balanced I/O scheduling greatly impacts writes than read workloads.

Table IV and Table V show the number of events processed by each core for OctoKV and OctoKV-LB, respectively. We considered each BDEV as a separate event. Threads are essentially executed sequentially for NVMe-oF target BDEV, KV store BDEV, and NVMe driver BDEV regarding I/O. However, through message passing, some BDEV operations of I/O can be handled by different cores. The original client requested a total of 360,000 NVMe commands, but for convenience of analysis, we only analyzed the number of processed events for the 100,000 consecutive I/Os for every core. Also, note that three events are generated for each NVMe command, resulting in a total of 300,000 events across both tables. In our experiment, the total of 100,000 I/O requests are transformed into a total of 300,000 events. As shown in Table IV, Cores #1&#2 process more NVMe command events than the other cores, indicating an overload on Cores #1&#2. Therefore, as shown in Table V, OctoKV-LB distributes the KV store and NVMe driver events from Core #1&#2 to the other cores.

TABLE VI
COMPARISON OF SCHEDULING HEURISTICS.

Workload		Medium Workload		Heavy Workload	
		Latency (µs)	Throughput (MB/s)	Latency (us)	Throughput (MB/s)
Fill Random	RR	247	617	278	663
	PS	244	630	274	680
Read Random	RR	325	613	342	720
	PS	325	615	342	724



(a) Medium Workload (Put) (b) Heavy Workload (Put)

Fig. 8. Time series analysis of OctoKV-LB for medium and heavy workloads with "Fill Random" using T_{OL}=0.4, T_{LB}=0.1, and a PS algorithm.

This way, OctoKV-LB resolves the core overloading problem on Core #1&#2.

2) Comparative Analysis of Scheduling Heuristics: In the previous experiments, we only showed the results for a PS algorithm. Here, we compare the RR and PS scheduling heuristics with OctoKV-LB. Table VI shows the results. Overall, PS achieved slightly higher or equivalent throughput compared to RR in all cases. As for I/O response time, PS tends to be slightly lower than RR for Put workloads, while the two are nearly identical for Get workloads. Therefore, it can be inferred that the benefits of load balancing outweigh the computational overhead incurred in scheduling decision-making.

3) Time Series Analysis: Figure 8 presents the time series results of OctoKV-LB's core utilization for medium and heavy workloads with the "Fill Random" option. The dotted line in the figure (indicated at 15 seconds) represents the point at which load-balancing I/O scheduling is triggered. In other words, before the dotted line, load-balancing I/O scheduling was not performed, and after the dotted line, it is performed.

Figure 8(a) is the results for medium workload. Before the 15-second mark, there is a significant deviation in core utilization between the high and low groups. Once the system triggers load balancing after 15 seconds, I/O events (KV store and NVMe driver) in the high group are redistributed to the low group, resulting in a noticeable shift in core utilization patterns. Specifically, we observe a decrease in core utilization for the high group and a corresponding increase in core utilization for the low group. This suggests that the load balancing mechanism is effective in evenly distributing the workload across both groups. And, as explained earlier, NVMe-oF consumes a significant amount of CPU cycles, so the core utilization of the high and low groups in Figure 8 does not converge to the same value. When all events of the cores in the high group are migrated, their core utilization decreases by approximately 20% to the maximum.

Figure 8(b) shows the results for heavy workload. Notably, we observe an overall increase in core utilization across all cores compared to Figure 8(a). Additionally, similar to the findings presented in Figure 8(a), our visual analysis suggests that the load balancing mechanism operates efficiently between cores following the 15-second mark.

4) Sensitivity Analysis: The effectiveness of load balancing in OctoKV-LB depends on the setting of the threshold values (T_{OL}, T_{LB}). Therefore, we evaluated the throughput of OctoKV-LB for heavy workload (Put) for various combinations of threshold settings (T_{OL}, T_{LB}). The default setting we have used so far is (0.4, 0.1). When we adjusted the thresholds to (0.7, 0.4), there was no change in throughput. In contrast, if we set T_{OL} or T_{LB} to too high values, load balancing does not work properly. Therefore, (0.9, 0.1) and (0.4, 0.6) showed a low throughput because load balancing did not work properly.

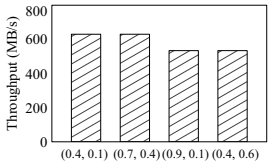


Fig. 9. The impact of (T_{OL}, T_{LB}) settings on throughput.

VI. CONCLUSION

In this paper, we proposed OctoKV, a network-based server-side KV store that leverages the SPDK capabilities for high performance in disaggregated storage. We identified and showed experimentally significant load imbalance in SPDK-based KVS. We proposed a robust load-aware balanced I/O scheduling in OctoKV to cater the load imbalance problem. The evaluation shows that OctoKV achieves lower I/O response times in comparison to traditional approaches where key-value stores run on the client side. Furthermore, the proposed load balancing strategies effectively optimize I/O response time by evenly distributing the load across the cores.

ACKNOWLEDGMENTS

We thank the anonymous reviewers of our paper for their invaluable comments on improving the paper. This work was funded in part by the Institute of Information Communications Technology Planning Evaluation (IITP) grants funded by the Korea government (MSIT) (No. 2020-0-00104), in part by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. NRF-2021R1A2C2014386), and in part by SK hynix research grant. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] H. Lim, D. H. Han, D. G. Anderson, and M. Kaminsky, "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage," in *USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2014.

[2] J. Yang, Y. Y. Yue, and K. V. Rashmi, "A Large Scale Analysis of Hundreds of In-Memory Cache Clusters at Twitter," in *USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2020.

[3] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *ACM Symposium on Operating Systems Principles*, SOSP, 2017.

[4] C.-G. Lee, H. Kang, D. Park, S. Park, Y. Kim, J. Noh, W. Chung, and K. Park, "iLSM-SSD: An Intelligent LSM-Tree Based Key-Value SSD for Data Analytics," in *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2019.

[5] J. Im, J. Bae, C. Chung, Arvind, and S. Lee, "PinK: High-speed In-storage Key-value Store with Bounded Tails," in *USENIX Annual Technical Conference (ATC)*, 2020.

[6] SK hynix and Los Alamos National Laboratory, "Los Alamos National Laboratory and SK hynix to demonstrate first-of-a-kind ordered Key-value Store Computational Storage Device." https://discover.lanl.gov/news/0728-storage-device, 2022.

[7] Samsung Electronics Co., Ltd., "Samsung Successfully Develops Industry's First 'Key Value' SSD Prototype that Meets New Open Standard." https://news.samsung.com/us/samsung-successfully-develops-industrys-first-key-value\-ssd-prototype-meets-new-open-standard/, 2019.

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

[9] H. Chen, C. Ruan, and C. Li, "SpanDB: A Fast, Cost-Effective LSM-Tree Based KV Store on Hybrid Storage," in *USENIX Conference on File and Storage Technologies*, 2021.

[10] T. Yoshimura, T. Chiba, and H. Horii, "EvFS: User-level, Event-Driven File System for Non-Volatile Memory," in *USENIX Annual Technical Conference (ATC)*, 2019.

[11] K. Lu, N. Zhao, and J. Wan, "TridentKV: A Read-Optimized LSM-Tree Based KV Store via Adaptive Indexing and Space-Efficient Partitioning," *IEEE Transactions on Parallel and Distributed Systems*, Aug 2022.

[12] Intel, "Message passing and concurrency." https://spdk.io/doc/concurrency.html, 2019.