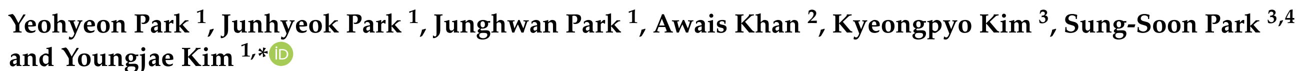
electronics

MDPI

Article

OctoFAS: A Two-Level Fair Scheduler that Increases Fairness in Network-Based Key-Value Storage


Yeohyeon Park [1], Junhyeok Park [1], Junghwan Park [1], Awais Khan [2], Kyeongpyo Kim [3], Sung-Soon Park [3,4] and Youngjae Kim [1,*]

[1] Department of Computer Science and Engineering, Sogang University, Seoul 04107, Republic of Korea; yeohyeon@sogang.ac.kr (Y.P.); junttang@sogang.ac.kr (J.P.); junghwan@sogang.ac.kr (J.P.)
[2] Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA; khana@ornl.gov
[3] GlueSys Co., Ltd., Anyang 14055, Republic of Korea; kpkim@gluesys.com (K.K.); sspark@gluesys.com (S.-S.P.)
[4] Department of Computer Science and Engineering, Anyang University, Anyang 14028, Republic of Korea
* Correspondence: youkim@sogang.ac.kr; Tel.: +82-2-705-8933

Abstract: We identified a fairness problem in a network-based key-value storage system using Intel Storage Performance Development Kit (SPDK) in a multitenant environment. In such an environment, each tenant's I/O service rate is not fairly guaranteed compared to that of other tenants. To address the fairness problem, we propose OctoFAS, a two-level fair scheduler designed to improve overall throughput and fairness among tenants. The two-level scheduler of OctoFAS consists of (i) inter-core scheduling and (ii) intra-core scheduling. Through inter-core scheduling, OctoFAS addresses the load imbalance problem that is inherent in SPDK on the storage server by dynamically migrating I/O requests from overloaded cores to underloaded cores, thereby increasing overall throughput. Intra-core scheduling prioritizes handling requests from starving tenants over well-fed tenants within core-specific event queues to ensure fair I/O services among multiple tenants. OctoFAS is deployed on a Linux cluster with SPDK. Through extensive evaluations, we found that OctoFAS ensures that the total system throughput remains high and balanced, while enhancing fairness by approximately 10% compared to the baseline, when both scheduling levels operate in a hybrid fashion.

Keywords: high-performance I/O; key-value store; storage system; I/O scheduling

Citation: Park, Y.; Park, J.; Park, J.; Khan, A.; Kim, K.; Park, S.-S.;Kim, Y. OctoFAS: A Two-Level Fair Scheduler that Increases Fairness in Network-Based Key-Value Storage. *Electronics* **2024**, *13*, 619. https://doi.org/10.3390/electronics13030619

Academic Editor: Fernando De la Prieta Pintado

Received: 24 November 2023
Revised: 26 January 2024
Accepted: 31 January 2024
Published: 1 February 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Embedded key-value storage (KVS) systems like LevelDB [1] and RocksDB [2] have gained widespread use as storage solutions for data-intensive applications. Traditionally, these stores run on top of file systems, which involves the conversion of `<key, value>` pairs into file and then into block addresses, incurring a significant performance overhead during I/O operations [3]. Additionally, running a KVS on top of a file system introduces additional overhead in the form of operating system operations, such as user-kernel mode switching, interrupt handling, and context switching. With the advent of ultra-low-latency storage devices like NVMe SSDs [4–6], these overheads have become worse [7].

To address this issue, various methodologies have been introduced to eliminate the file system and operating system components in the I/O path of KVS [8–12]. One prominent approach is the Intel Storage Performance Development Kit (SPDK) [13]. The SPDK utilizes a user-level NVMe driver with polled mode, which plays a significant role in reducing the performance overhead traditionally associated with file and operating systems. It reduces context switching between user and kernel modes, repetitive block address translation, and eliminates costly interrupts. Due to these characteristics, the SPDK can fully leverage the true capabilities of NVMe SSDs. Consequently, there are ongoing efforts to integrate the SPDK into the development of high-performance KVS solutions [14–16].

On the other hand, in contemporary computing environments, the trend of storage disaggregation is on the rise [17–20]. Storage disaggregation, achieved through storage virtualization, allows remote disks or file systems to be presented as if they were local. This method is commonly employed in cloud services like Amazon's EBS and EFS, which provide block and file storage capabilities, respectively. In such disaggregated storage scenarios, the conventional methodology of developing embedded KVSs still requires address translation at both the file and block storage layers. This process involves additional overhead due to context switching within the kernel.

To address this issue, a network-connected disaggregated key-value storage system called OctoKV [21] has been recently introduced. In OctoKV, the client directly transmits key-value requests to the server, bypassing the file system or KVS. OctoKV then implements the key-value storage engine on the user-level NVMe driver on the server using Intel SPDK and exports it to the client over the network. This approach allows OctoKV to bypass all the block and file components that contribute to the aforementioned overhead on both the client and server sides, thereby leading to a substantial improvement in performance. Furthermore, OctoKV leverages SPDK's support for NVMe over Fabrics (NVMe-oF), enabling the development of disaggregated storage architectures that offer improved performance and scalability.

However, despite its current design, we have identified a key limitation in OctoKV: the lack of support for multiple tenancy, an essential feature in cloud shared-storage environments. In such environments, where multiple tenants share resources, each tenant expects to receive fair I/O service, but OctoKV does not guarantee fair I/O bandwidth allocation to each tenant. Within OctoKV's SPDK framework, each core sequentially processes the TCP socket queue and the event queue. Key-value I/O requests from tenants are first placed into the socket queue. These requests are then converted into various events and fed into the event queue, where they are processed in the order they were received. If I/O requests from more than two tenants are concurrently inserted into the sequence of these queues, but with varying intensities, interference can occur in the I/O service between them, potentially leading to unfairness in the I/O service experienced by each tenant.

In this paper, we introduce OctoFAS as a solution to address this multi-tenancy problem in OctoKV. OctoFAS is an SPDK-based KVS equipped with a two-level fair scheduler featuring inter-core and intra-core scheduling to enhance the overall system performance and fairness simultaneously.

- Firstly, the inter-core scheduling dynamically manages the allocation of I/O requests from aggressive tenants. It performs runtime migrations of these requests from overloaded cores to underloaded cores, ensuring an even distribution of CPU utilization across multiple cores while also considering fairness as a critical factor.
- Secondly, OctoFAS incorporates intra-core scheduling, which gives a higher priority to I/O requests from starving tenants over those from well-fed tenants within the per-core event queue,to deliver a fair key-value I/O service to multiple tenants.

Importantly, these two policies can be dynamically enabled or disabled based on runtime analysis of each tenant's latency and each core's CPU utilization, allowing OctoFAS to respond effectively to various workloads.

We implemented OctoFAS based on SPDK v21.10 and conducted an evaluation using the RocksDB db_bench benchmark. Our experimental results indicate that OctoFAS maintains a high and balanced total system throughput. Moreover, it improves fairness by 10% compared to the baseline when both inter-core and intra-core scheduling levels function in a hybrid manner.

2. Background

In this section, we describe the effectiveness of the user-level NVMe driver (Section 2.1) and SPDK-based server-side key-value storage system (Section 2.2).

2.1. Event-Driven User-Level NVMe Drivers

The Intel Storage Performance Development Kit (SPDK) [13] is one of the most popular storage frameworks for constructing high-performance storage systems, particularly with ultra-low-latency NVMe SSDs [4–6]. To achieve low latency and high throughput, SPDK provides a user-mode NVMe driver with a polled mode. This driver plays a crucial role in minimizing additional latency in the I/O response time of fast SSDs. This extra latency occurs when the SSD's own latency is shorter than the time required for interrupt handling and completion retrieval by the operating system. Furthermore, SPDK utilizes a per-core lockless event framework. This framework effectively mitigates potential lock contention issues that may arise from shared memory among CPU cores, resulting in optimal performance. Consequently, client I/O requests are intricately bound to a single core. Each request is seamlessly divided into multiple events, which are meticulously processed in a sequential manner by the designated core. This approach allows SPDK to fully leverage the performance potential of modern NVMe SSDs while maintaining low latency.

In SPDK, block device (BDEV) plays a fundamental role as a core component of the SPDK block device layer. Every event within the system must be processed through the BDEV. BDEV offers a pluggable module API that allows for the implementation of block devices that interface with block storage devices. Users have the flexibility to choose from existing BDEV modules or create virtual BDEV (VBDEV) modules. VBDEV modules enable the creation of block devices on top of existing BDEVs. These BDEV operations collectively handle comprehensive I/O requests. Using the VBDEV, SPDK can generate events for other BDEVs. Users have the freedom to customize VBDEV as a user-defined function and insert it within the I/O path to perform specific operations such as compression or encryption during I/O processing. In SPDK, each core operates a dedicated thread known as a reactor, which sequentially processes multiple pollers (functions).

2.2. SPDK-Based Server-Side Key-Value Storage System

NVMe over Fabrics (NVMe-oF) is a communication protocol designed to connect host systems to storage resources across a network fabric by utilizing the NVMe protocol as its foundation. NVMe-oF facilitates the transfer of data between a host system and a target storage device through the use of NVMe commands. The data associated with each NVMe command are transmitted over networks such as Ethernet, Fibre Channel (FC), or InfiniBand. Importantly, NVMe-oF enables users to interact with remote NVMe SSDs as if they are local storage devices, thus allowing for a fundamental form of storage disaggregation. SPDK extends support for NVMe-oF. It offers compatibility with both RDMA and TCP transports, providing NVMe-oF hosts tailored for client-side NVMe drivers and targets for the server side. In a typical KVS scenario, the KVS runs on a file system that can be either locally hosted or network-attached block storage. On the server side, SPDK runs an NVMe-oF target BDEV along with an NVMe driver BDEV to effectively manage incoming I/O requests from clients. The NVMe-oF target BDEV receives these client-initiated NVMe commands through designated socket queues, while the NVMe driver BDEV is responsible for generating block-level I/O operations and delivering them to the SSD for execution.

Figure 1 illustrates the software design of the server-side SPDK-based key-value storage system, OctoKV, and the event handling process within SPDK. OctoKV leverages SPDK's robust capabilities to support disaggregated storage architectures, effectively eliminating the kernel overhead associated with operating on top of the operating system and eliminating the file system overhead related to multiple address conversions. Specifically, Figure 1a depicts OctoKV's software stack. In OctoKV, the key-value storage engine operates within the SPDK framework on the server, streamlining the I/O stack on the client side by entirely bypassing the file system. OctoKV maintains the same key-value API as before, allowing applications running on the client side to access the key-value storage without requiring any modifications. Additionally, OctoKV provides a key-value API library with operations such as Put() and Get(), which are built on the NVMe protocol. On the server

side, SPDK manages the NVMe-oF target BDEV, key-value store (engine) BDEV, and NVMe driver BDEV to handle I/O requests from clients. The NVMe-oF target BDEV receives the client's NVMe commands through socket queues. The key-value store (engine) BDEV implements data structures like LSM-trees and Hash to manage key-value pairs efficiently. Meanwhile, the NVMe driver BDEV generates block-level I/O operations and delivers them to the SSD for execution.

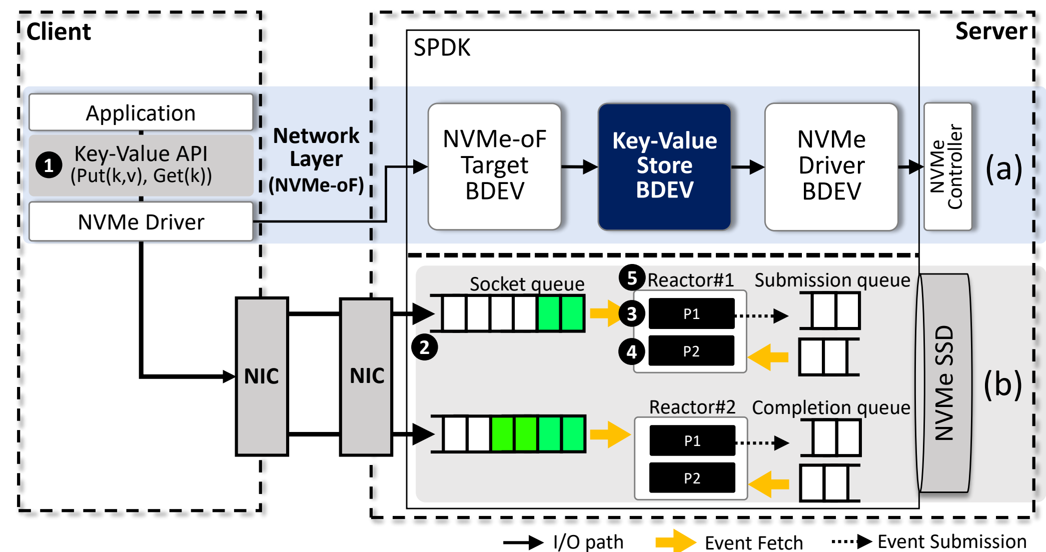


Figure 1. SPDK-based server-side key-value storage system [21]. (**a**) Software stack; (**b**) SPDK's event handling procedure.

Figure 1b demonstrates how the server-side SPDK processes I/O requests received from the client. The reactor, bound to the SPDK core, employs two pollers (functions) in a round-robin fashion to poll the queues essential for I/O processing. Specifically, the reactor executes Poller1 (P1) to poll the socket queue where NVMe commands are stored. It then proceeds to Poller2 (P2) to poll the completion queue where completed I/O operations from the SSD are recorded. The following describes the procedure through which the reactor leverages these pollers to manage the client's key-value I/O requests (write).

➊ The client initiates an I/O request to the server using Put(k, v). The client's NVMe driver then sends the NVMe command (Put) to the server's NVMe-oF target using the NVMe-oF protocol. ➋ After transmission, the NVMe command sent by the client is stored in the socket queue of the NVMe-oF target. Subsequently, SPDK's reactor executes the pollers in sequential order. ➌ The reactor executes poller (P1). Firstly, P1 first checks the socket queue, and, if it is not empty, P1 proceeds to dequeue an item from the queue. The item represents the NVMe command that was initially transmitted by the client. P1 then transforms this dequeued item into an SPDK event. Secondly, P1 sequentially executes the NVMe-oF target BDEV, key-value store (V)BDEV, and NVMe driver BDEV for the NVMe command. The NVMe-oF target BDEV converts the NVMe command received from the client into the SPDK's I/O format (bdev_io). After that, the key-value store BDEV takes the bdev_io as input and searches for the location (LBAs) of the value for the key. If the key-value store BDEV is based on a hash data structure, it executes a hash function for the key to find the location of the value. Then, the NVMe driver BDEV converts the bdev_io into block I/O and enqueues it into the submission queue. ➍ The reactor then executes the next poller (P2). P2 checks the completion queue of the NVMe driver to see if there are any completed I/O operations from the SSD. If there are completed operations, it dequeues and processes them. If the queue is empty, the reactor returns to executing P1 (➌). Otherwise, P2 notifies the NVMe driver BDEV, key-value store (V)BDEV, and NVMe-oF target BDEV through callbacks, indicating that the I/O operations have been successfully

completed. ❺ The reactor continues to repeat the above steps (❷ and ❸) to process the events accumulated in each queue.

Notably, OctoKV identifies the load imbalance problem within SPDK-based KVS. As depicted in Figure 2a, SPDK-based KVS exhibits an imbalance in workload distribution among TCP connections when clients send NVMe commands over TCP to the server. Despite clients establishing multiple TCP connections and the server running corresponding SPDK threads to process these commands, the distribution of workloads among connections remains uneven. This imbalance results in certain connections receiving a disproportionately heavier workload, incurring delays in I/O response times. Moreover, running the key-value storage engine on the server contributes to an increase in CPU load, which further intensifies the load imbalance issue. Particularly, Hash- or LSM-tree-based key-value storage engines introduce CPU-intensive tasks, such as executing hash functions and merge–sorts during compaction. This added CPU burden contributes to delays in response times, with the TCP layer remaining unaware of these application processes. Consequently, threads on connections experiencing high load significantly impact the overall response time. OctoKV effectively addresses these load imbalance problems by implementing an I/O event migration scheme that evenly distributes the I/O load across SPDK cores, thereby achieving load-aware fine-grained scheduling (see Figure 2b). OctoKV monitors the current core load distribution status and initiates online migrations when load imbalances are detected. These migrations transfer the load from high- to low-load core groups, resulting in a more balanced distribution of core loads across the server. Ultimately, this approach maximizes the throughput of the entire key-value storage system. The details regarding the migration scheme are further explained in Section 4.4.

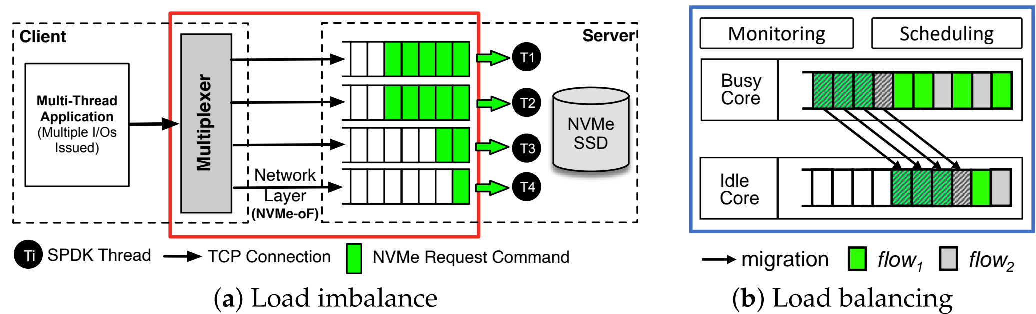


Figure 2. Visualizations of (**a**) load imbalance problem (red box) in SPDK-based KVS and (**b**) load-aware balanced I/O scheduling (blue box) to address the problem.

3. Motivation

Although the state-of-the-art SPDK-based KVS is optimized for throughput by load balancing, we confirmed that it does not guarantee fair I/O bandwidth allocation to each tenant (Section 3.2), which is a crucial aspect in cloud environments (Section 3.1). Moreover, we identified that the load imbalance not only affects the system performance but also the fairness (Section 3.3).

3.1. Fairness of I/O Services

In a system where multiple clients/tenants share storage, fairness is achieved when all clients receive an equal share of bandwidth, typically set at 1/N [22–34]. In other words, if the I/O operations of multiple clients are processed simultaneously and each client experiences an equal reduction in performance, the system is considered fair.

In our system, let us consider that one tenant corresponds to one I/O flow. For each I/O flow, denoted as $flow_i$, when it runs in isolation, its average response time is RT_i^{Alone}. When $flow_i$ runs in parallel with other tenants, its average response time is RT_i^{Shared}. Here, the response time of each I/O request is defined as the time it takes for the request to be initiated by the client and receive a response from the server. In this context, the slowdown of $flow_i$ (S_i) is defined as follows:

$$Slowdown\ (S_i) = \frac{RT_i^{Shared}}{RT_i^{Alone}} \tag{1}$$

Furthermore, in this scenario, the fairness of the system (F) is defined by measuring the ratio between the maximum and minimum slowdown observed across all flows in the system.

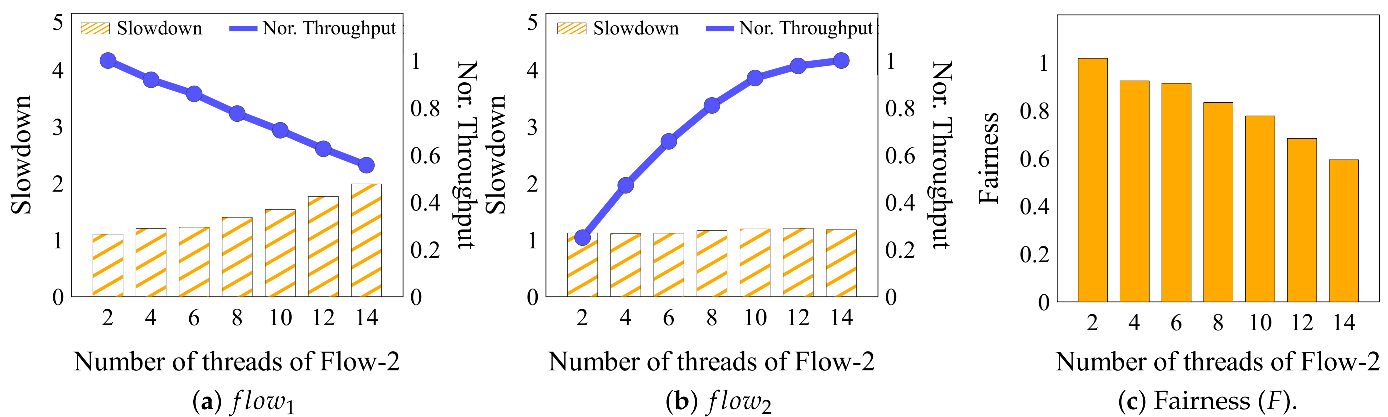
$$Fairness\ (F) = \frac{\min_i \{S_i\}}{\max_i \{S_i\}} \tag{2}$$

A smaller value of F indicates a greater difference between the flow that experiences the most significant slowdown and the flow with the least slowdown, signifying an unfair system. Conversely, a value of F closer to one signifies a fair system.

3.2. Experimental Analysis

To confirm that SPDK-based KVS does not provide or guarantee fairness to multiple clients/tenants in a multi-tenant environment, we conducted the following experiment; In the system, two different tenants ($flow_1$ and $flow_2$) with varying I/O workloads shared OctoKV for write operations. $flow_1$ employed two I/O threads to perform write operations, while $flow_2$ initiated write operations with a variable number of threads, ranging from a minimum of 2 to a maximum of 14, in increments of 2. This allowed for an increase in the I/O workload for $flow_2$. Details regarding the I/O load for each tenant and the experimental environment are described in Section 5.

Figure 3 shows the experimental results, showing the throughput and slowdown of each flow. In Figure 3a, as the I/O workload of $flow_2$ increases, the slowdown of $flow_1$ gradually increases from 1 to 2, while its normalized throughput decreases to 0.55. This occurs because $flow_2$ requests a significant amount of I/O, which interferes with the execution of $flow_1$, leading to increased latency for $flow_1$. Conversely, the performance of $flow_2$ in Figure 3b demonstrates that the slowdown values remain almost unchanged, and throughput values are continuously increasing.



(**a**) $flow_1$ (**b**) $flow_2$ (**c**) Fairness (F).

Figure 3. Slowdown and normalized throughput of each I/O flow, $flow_1$ and $flow_2$, when they concurrently use SPDK-based KVS, and the resulting fairness.

In accordance with the slowdown values of the two I/O flows, fairness is also depicted using Equation (2). As shown in Figure 3c, when both I/O flows have a thread count of 2, the fairness value is mostly equal to 1, which is the expected level of fairness. However, as the thread count of $flow_2$ increases, the fairness values sharply decrease. This experiment highlights that SPDK-based KVS encounters an unfairness problem when multiple tenants send varying amounts of requests concurrently. Considering the ultimate goal of SPDK-based KVS in a cloud environment with multiple tenants, it is essential to address

the fairness issue. Without ensuring fairness, even if overall latency and throughput are excellent, it can pose significant problems.

3.3. Correlation between Load Imbalance and Fairness

We discovered that there is a correlation between the unfairness among clients and the I/O request processing capacity per SPDK core. SPDK establishes TCP channels between clients and the server using NVMe-oF, and the number of TCP connections is equal to the number of SPDK cores on the server. Each client's transport layer sends I/O requests to the server via TCP connections. When doing so, it sends a certain amount of requests that do not exceed a threshold on the first TCP connection and then sends the remaining requests on subsequent TCP connections. Consequently, the amount of I/O requested per TCP connection may vary, leading to different numbers of NVMe commands accumulating in the server's socket queue for each core. Furthermore, the NVMe commands in the socket queue may consist of requests from various tenants.

In Figure 4, to conduct a close examination, we analyzed the correlation between I/O latency and CPU core utilization on the server side of SPDK-based KVS. It depicts the results of I/O latency in SPDK concerning server core utilization. The baseline SPDK-based KVS utilizes six CPU cores. To gain a clear understanding of the fairness issue, we conducted experiments for the scenario where fairness falls below 0.8. Specifically, we used 2 threads for $flow_1$ and 10 threads for $flow_2$. Note that the I/O latency mentioned here is not the response time from the client's perspective but rather a value measured on the server side.

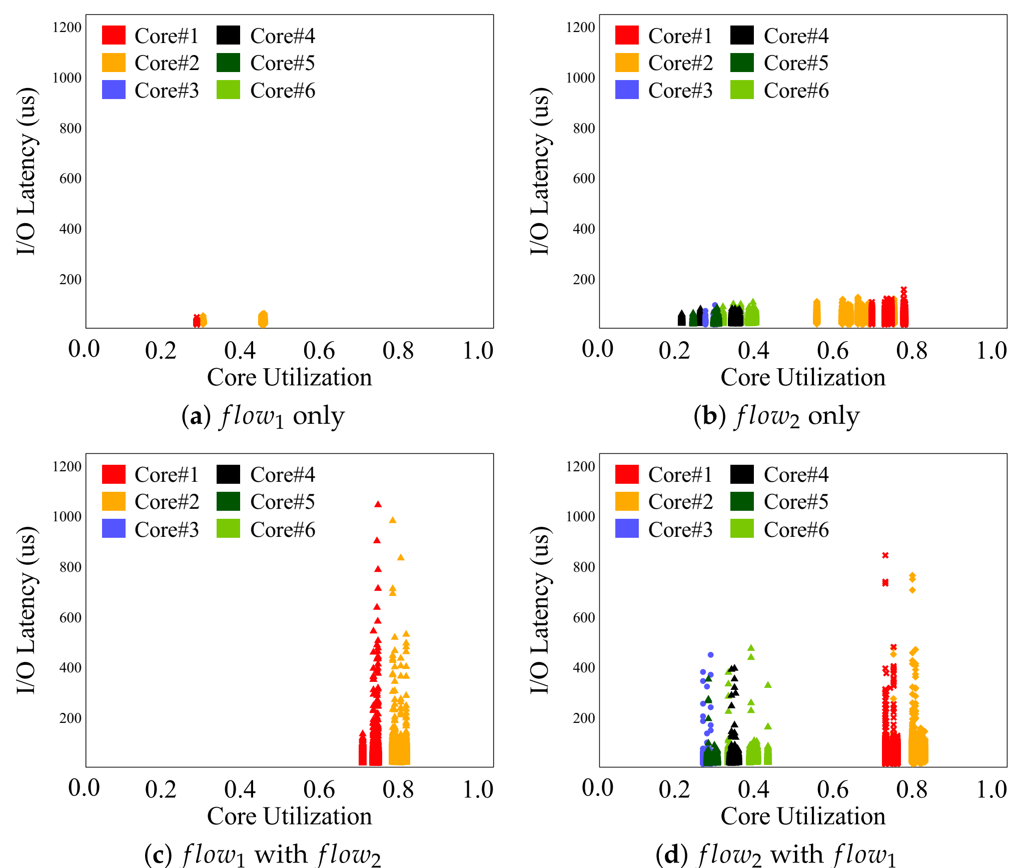


Figure 4. (**a**,**b**) I/O latency and CPU utilization of $flow_1$ and $flow_2$ when executed individually. (**c**,**d**) Results of simultaneously running $flow_1$ and $flow_2$.

Figure 4a,b represent the results when $flow_1$ and $flow_2$ are processed individually. In Figure 4a, the server utilizes only two out of the six cores for I/O processing. On the other hand, when $flow_2$ with higher I/O demands than $flow_1$ is processed, as observed in Figure 4b, all six cores are engaged in processing $flow_2$. At this point, cores#1 and#2 exhibit a core utilization of 0.6 or higher, while the utilization of the remaining cores is 0.4 or lower, resulting in a bimodal distribution of core utilization. As the I/O volume increases, there is a noticeable load imbalance among the cores, and high-core-utilization cores experience longer latencies. On the other hand, Figure 4c,d present the results for $flow_1$ and $flow_2$ when executed simultaneously. Figure 4c demonstrates a significant increase in latency for $flow_1$ when executed concurrently. In particular, $flow_1$ is processed on overloaded core#1 and core#2 due to the presence of $flow_2$, resulting in a substantial increase in I/O latency. Similarly, in Figure 4d, the increased I/O from $flow_2$ leads to load imbalance, causing an increase in I/O latency. However, the magnitude of the latency increase is relatively smaller compared to the increase in $flow_1$.

In summary, the load imbalance situation induced by $flow_2$ has a detrimental impact on fairness. When load imbalance occurs due to $flow_2$, $flow_1$ is processed on cores with high utilization, leading to a significant increase in latency. Thus, load imbalance not only affects throughput but also has a negative impact on fairness. We propose OctoFAS as a solution to address the challenges involving a mix of fairness and load imbalance.

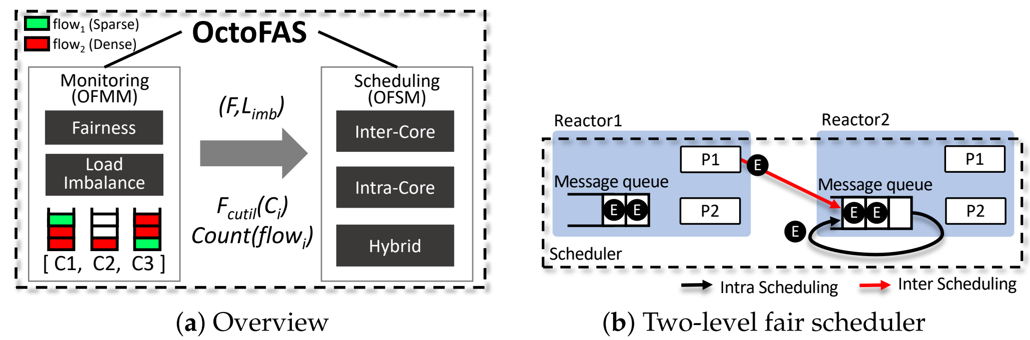
4. Design and Implementation

In this section, we explain the components of OctoFAS. We describe the overall system structure (Section 4.1), a monitoring module that monitors fairness and core utilization at fixed time intervals (Section 4.2), a scheduling module equipped with a two-level fair scheduler (Section 4.3), and implementation details including operational flows (Section 4.4).

4.1. System Overview

As confirmed in Section 3, SPDK-based KVS fails to ensure fairness among I/O flows. Additionally, the load imbalance problem originating from SPDK not only reduces throughput but also has a negative impact on fairness among I/O flows. To address this combination of issues, we present OctoFAS, an innovative SPDK-based network key-value storage system with a two-level I/O event scheduling technique. The goal was to enhance overall server throughput through improved load balancing and ensure fairness by changing the order of I/O processing.

Figure 5a provides an overview of the OctoFAS system architecture. OctoFAS consists of the OctoFAS Monitoring Module (OFMM) and the OctoFAS Scheduling Module (OFSM). OctoFAS was designed to adapt to dynamically changing workloads based on the integrated operation of OFMM and OFSM. OFMM monitors for unfairness and load imbalance issues among I/O flows at fixed time intervals. OFSM, in turn, makes scheduling decisions based on the information collected by OFMM and employs a robust scheduling algorithm. Figure 5b illustrates the high-level operation of the two-level fair scheduler proposed in OctoFAS. The scheduling in the two-level fair scheduler is divided into inter-core scheduling and intra-core scheduling. Inter-core scheduling, similar to the load-aware balanced I/O scheduling proposed in traditional SPDK-based KVS [21], aims to increase throughput but with different migration targets. Intra-core scheduling operates within each core's event queue and enhances fairness. These two scheduling mechanisms can be independently turned on or off and can operate concurrently in a hybrid fashion. Further details are discussed in the following paragraphs.



(**a**) Overview (**b**) Two-level fair scheduler

Figure 5. System overview of OctoFAS featuring a two-level fair scheduler.

4.2. OctoFAS Monitoring Module (OFMM)

OFMM monitors the slowdown values of each I/O flow and the core utilization of each core within a fixed time interval (W). This fixed time interval is an adjustable parameter, allowing flexibility in the monitoring process.

4.2.1. Fairness Monitoring

As seen in Equations (1) and (2), the slowdown and fairness, respectively, of each flow are calculated based on the average I/O latency of each I/O flow. The average I/O latency of each I/O flow is denoted as $f_{latency}(flow_i)$, where $flow_i$ ranges from $flow_1$ to $flow_t$. A $flow$ represents a group consisting of t flows, denoted as $flow_1$, $flow_2$, and so on. However, it is challenging to track the actual I/O latency experienced by remote clients within the SPDK running on the server. Therefore, OctoFAS estimates the actual I/O latency based on the measured I/O latency within SPDK. OctoFAS corrects the values using the difference between the I/O latency measured within SPDK and the I/O latency measured by clients. The client I/O latency, denoted as RT_i, is composed of the network round-trip time T_{Net} and $f_{latency}(flow_i)$, which can be expressed as $RT_i = T_{Net} + f_{latency}(flow_i)$. When T_{Net} occupies a large portion of RT_i, the fairness calculated using $f_{latency}(flow_i)$ is significantly reduced. Hence, we added the measured T_{Net} to $f_{latency}(flow_i)$ to calculate the fairness. Once the calculated fairness based on the measured slowdown values is 0.9 or lower, OFMM sets the unfairness flag. Through this process, OFMM determines the occurrence of unfairness.

4.2.2. Core Utilization Monitoring

OFMM periodically collects core utilization information for each core using the APIs provided by SPDK. Core utilization is denoted as $f_{cutil}(c_i)$, where c_i ranges from c_1 to c_n. Here, C represents a core group consisting of n cores, denoted as c_1, c_2, and so on. OFMM checks whether load imbalance exists among the cores by examining the measured core utilizations and using the following equation:

$$LoadBalance\ (LB) = \frac{\min_i \{f_{cutil}(c_i)\}}{\max_i \{f_{cutil}(c_i)\}} \tag{3}$$

The equation provided calculates the load imbalance by measuring the ratio between the maximum and minimum core-utilization values of all cores. This logic is similar to the one used for calculating fairness. Balanced load can be seen as a scenario where the core utilization is evenly distributed, and OFMM compares this load imbalance value (LB) with a predefined threshold, denoted as T_{LB}, which ranges from 0.0 to 1.0. The user sets T_{LB}, and it should be configured to match the specific system environment. If LB is lower than T_{LB}, OFMM sets the load imbalance flag. Subsequently, OFMM divides the cores into two groups based on their utilization: one group with high core utilization and another with low core utilization. The average utilization of all cores, denoted as U_{avg}, is used as the criterion to differentiate between these two groups. OFMM passes these flag values and information about the two groups and U_{avg} to OFSM, enabling it to trigger the appropriate scheduling to tackle the load imbalance and improve system performance and fairness.

4.3. OctoFAS Scheduling Module (OFSM)

As described earlier, OFMM evaluates fairness values and load imbalance for each time interval (W) to select the appropriate scheduling policy among the three available. OFSM, as depicted in Algorithm 1, classifies four cases based on the truth table for the two flags received from OFMM and determines the scheduling policy it will perform. For these cases, excluding (F, F), OFSM provides three algorithms as outlined below (see Algorithm 1).

Algorithm 1 OctoFAS Scheduler Module (OFSM).

```
1:  /* OctoFAS Scheduler Module */
2:  while EventQ is not empty do
3:      E_head ← Event retrieved from the head of EventQ
4:      if unfairness is F and loadImbalance is F then
5:          FORWARDTOKVS(E_head)
6:      else if unfairness is F and loadImbalance is T then
7:          INTERSCHEDULE(Core, E_head)
8:      else if unfairness is T and loadImbalance is F then
9:          INTRASCHEDULE(Flow, Core, E_head)
10:     else if unfairness is T and loadImbalance is T then
11:         HYBRIDSCHEDULE(Flow, Core, E_head)
```

4.3.1. Inter-Core Scheduling

Inter-core scheduling aims to address the load imbalance problem by migrating events from overloaded cores to underloaded cores, thus achieving a fair queue depth across all cores' event queues. This algorithm operates similarly to the load-aware balanced scheduling used in the previous SPDK-based KVS [21]. It leverages the message passing functionality provided by SPDK, and further details can be found in Section 4.4.2. The goal is to equalize the load on each core, ultimately increasing the overall server throughput.

In this situation, determining the number of I/O events to be migrated from the high- to the low-util group CPUs is a critical decision since moving too many I/O events could overload the cores in the low group. There are various approaches to making this decision, but the simplest and most reliable method is to migrate I/O events one by one in a round-robin order from each core in the high-utilization group to the cores in the low-utilization group sequentially. However, when the core utilization of the current target low-utilization-core approaches U_{avg} due to migration, that low-utilization core is temporarily excluded from migration. Similarly, when a high-utilization core's utilization approaches U_{avg} due to migration, migration is paused on that core. This straightforward approach helps balance the core loads and prevents situations where the overloaded core that is performing migrations becomes underloaded, while the core receiving migrations becomes overloaded. This can be clearly confirmed in Algorithm 2.

Algorithm 2 OFSM inter-core scheduling.

```
1:  function INTERSCHEDULE(Core, E_head)
2:      Core_temp ← an arbitrary core of C_low (starting point)
3:      if Core ∈ C_high and COREUTIL(Core) ≥ U_avg then
4:          while COREUTIL(Core_temp) ≥ U_avg do
5:              Core_temp ← the next core in C_low
6:          PUSHTOMSGQ(Core_temp, E_head, Flow)
7:          Core_temp ← the next core in C_low
8:          return T
9:      else
10:         return F
11:
12: /* Message queue poller for inter-core scheduling */
13: while MsgQ is not empty do
14:     E_head ← Event retrieved from the head of MsgQ
15:     FORWARDTOKVS(E_head)
```

Load balancing in OctoFAS differs from the load-aware scheduling of the former SPDK-based KVS in that it offers flexibility in deciding which flows of events, among those on overloaded cores, to migrate to other underloaded cores. In the SPDK-based KVS, all events on overloaded cores were migrated without considering the intensity of the requesting flows. This was because it was the most straightforward way to maximize overall throughput. However, OctoFAS provides system administrators with the freedom to choose which flow to migrate based on the intensity of each flow calculated during the fairness monitoring. This allows for the consideration of system-wide fairness. For example, if it is determined that migrating events from only the flows perceived as dense during an in-house evaluation in the target system can maintain high throughput and increase fairness, such a configuration is possible. In overloaded cores, there may be situations where it is deemed beneficial to remove events from dense flows to benefit sparse flows. We divided the decision on which flow's events to migrate into three cases and conducted detailed experiment. The results can be found in Section 5.2.1.

4.3.2. Intra-Core Scheduling

This algorithm is triggered when unfairness issues occur without the load imbalance in core utilization. It intentionally delays the processing order of events from dense flows within the queue through backwarding, which means fetching the request at the front of the queue and re-inserting it at the back, ensuring that events from sparse flows are processed first. The number of backwarding iterations is crucial and is determined according to the following formula:

$$I_{sparse_avg} = \frac{\sum_{i=1}^{n} f_{intensity}(flow^{i}_{low})}{n} \tag{4}$$

$$Backwardings(B) = \frac{f_{intensity}(flow^{i}_{high})}{I_{sparse_avg}} \alpha \tag{5}$$

In OFMM, the average intensity of low-intensity flows identified as sparse flows is calculated. Intensity per flow, denoted as $f_{intensity}$, is easily determined in OFSM by measuring the requests per second. The number of backwarding iterations is calculated by multiplying α with the result of dividing the intensity of flows in the dense flow group by the average intensity value of sparse flows. α is a tunable constant specific to the system environment. α should be set as a value that maximizes fairness while maintaining throughput at an acceptable level, as determined through in-house exploratory runs.

Once again, we conducted detailed case experiments on this, and the results can be found in Section 5.2.2. Once the number of backwarding iterations is determined, that number of backwarding iterations is applied to the events from dense flows in each event queue, rotating them idly to ensure that events from sparse flows are processed with high priority. In the memory space allocated to OFSM, an array with an entry for each connected tenant of OctoFAS is created. Each entry is used to record the backwarding setting value for the corresponding flow. Subsequently, for each core's message queue, a dynamic tracking variable is assigned to each event currently inserted in the queue to record how many times the event has been moved in and out within the queue. Comparing the number of rotations performed with the number of rotations needed ensures that the intentional delay occurs correctly. Detailed operations for this can be found in Algorithm 3.

Algorithm 3 OFSM intra-core scheduling.

```
1:  function IntraSchedule(Flow, Core, E_head)
2:      if Flow ∈ F_low then
3:          ForwardToKVS(E_head)
4:      else
5:          PushToMsgQ(Core, E_head, Flow)
6:
7:  /* Message queue poller for intra-core scheduling */
8:  while MsgQ is not empty do
9:      E_head ← Event retrieved from the head of MsgQ
10:     if Rotated[E_head] < RotateNum[Flow] then
11:         PushToMsgQ(Core, E_head, Flow)
12:         Rotated[E_head] = Rotated[E_head] + 1
13:     else
14:         ForwardToKVS(E_head)
```

Intra-core scheduling intentionally delays the processing order of dense flows to give a relative advantage to sparse flows. However, it is important to note that delaying dense flows incurs a certain amount of round-robin overhead. As a result, while overall system fairness increases, the overall throughput may decrease. Therefore, as mentioned earlier, the number of backwarding iterations should be thoughtfully configured.

4.3.3. Hybrid Scheduling

Hybrid scheduling is applied when both load imbalance of core utilization and unfairness issues arise simultaneously. In such cases, the two algorithms mentioned above operate in a two-level manner. First, through inter-core scheduling, I/O events are migrated from cores with high loads to cores with lower loads. Simultaneously, within each core's event queue, intra-core scheduling is applied, rearranging the order of I/O requests for dense flows to prioritize sparse flows (see Algorithm 4). Note that the message queue poller function is used in intra-core scheduling. For proper determination of the number of backwarding iterations during the intra-core scheduling, OFSM checks the flow of requests migrated to the core's queue and updates (adds) the count of requests for that flow.

Algorithm 4 OFSM hybrid scheduling.

```
1:  function HybridSchedule(Flow, Core, E_head)
2:      if Flow ∈ F_low then
3:          ForwardToKVS(E_head)
4:      else if InterSchedule(Core, E_head) is F then
5:          PushToMsgQ(Core, E_head, Flow)      /* Msg queue poller of intra-core scheduling */
```

4.4. Implementation

4.4.1. Hash-Based Key-Value Storage Engine

We implemented a KVS engine with SPDK's VBDEV module by adopting a hash-based data structure to prevent the excessive performance variations associated with LSM-tree compaction, allowing for a clearer observation of the scheduling impact.

4.4.2. Message Passing Based Two-Level Fair Scheduler

We focused on SPDK's message passing infrastructure [35] to achieve I/O reordering (scheduling). SPDK employs a message passing approach instead of a traditional locking method, enabling concurrency in multi-threaded programming and ensuring linear scalability with the addition of storage devices. Each reactor thread has its message queue, which facilitates communication between multiple reactors by exchanging events as messages in each other's message queues. As part of this mechanism, the reactor checks its message queue for pending messages between the execution of two pollers, P1 and P2 (refer to

Section 2.2). We utilized this feature of SPDK to implement the proposed two-level fair scheduling algorithms.

Instead, we made slight adjustments to the message queue poller routine of each reactor to execute the key-value storage engine VBDEV for migrated events, thus enabling the utilization of per-core message queues as channels for event migrations, as outlined in Algorithms 2 and 3. Subsequently, the reactor with high utilization pushes events to the message queue of the target low-utilization cores, taking into account the calculated number of event migrations (see Figures 5b and 6).

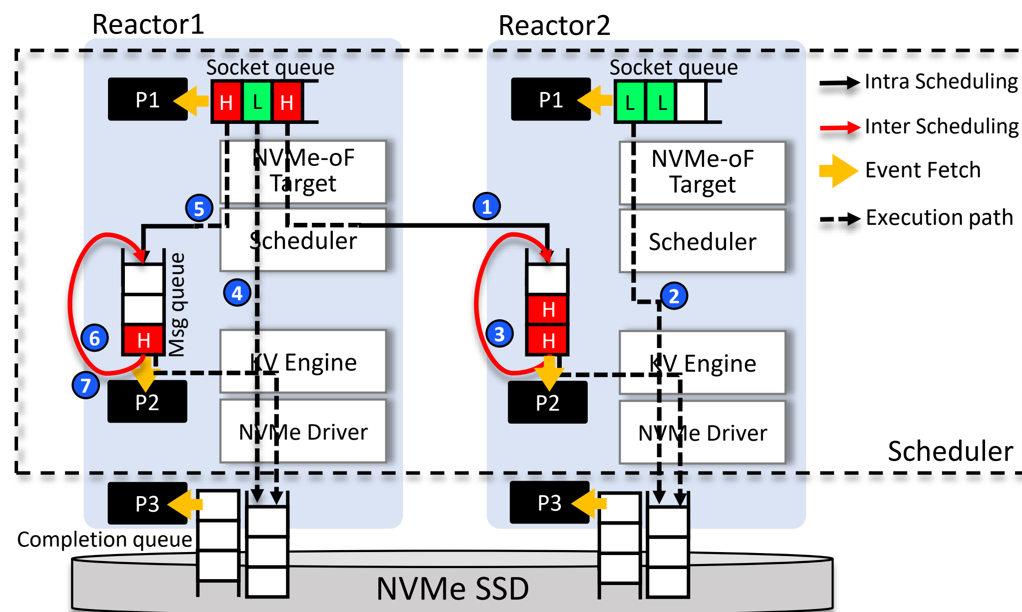


Figure 6. Operational flow when the hybrid scheduling is turned on by OctoFAS scheduling module.

4.4.3. Operational Flow

Figure 6 depicts the behavior of OFMM and OFSM using SPDK's BDEVs when an SPDK-based KVS runs on storage nodes. To provide a detailed understanding of the system's operation flow, let us consider a scenario where each I/O request enters the socket queue. Reactor 1 represents the thread of a core in the high group, while Reactor 2 corresponds to a core in the low group. I/O request events from dense intensity flows are denoted as *H*, and events from sparse intensity flows are denoted as *L*. OFMM monitors both unfairness and load imbalances, and, when it detects these issues, it reports them to OFSM and OFSM starts scheduling (hybrid scheduling is applied in this example). Note that the *j*th poller of each *i*th reactor is labeled as $R_i_P_j$.

❶ $R_1_P_1$ processes I/O requests from H that is in the socket queue. After executing the NVMe-oF target BDEV operation for this event, the I/O request is designated for inter-core scheduling in the OFSM BDEV and sent to Reactor2's message queue. Following this, R1 checks for KVS BDEV events, but since there are none, it proceeds directly to check $R_1_P_2$. However, the message queue is also empty, so it is skipped. R1 then proceeds to execute $R_1_P_3$, but as the completion queue is also empty, it returns to running $R_1_P_1$. ❷ Meanwhile, at this point, Reactor2 executes $R_2_P_1$, which handles the I/O from L in the socket queue. Since it is from L, this I/O is directly passed through the NVMe-oF Target BDEV, I/O scheduler BDEV, KV BDEV, and NVMe driver BDEV and delivered to the SSD. ❸ Next, Reactor2 proceeds to execute $R_2_P_2$. Here, it checks the message queue and examines the H event previously migrated from $R_2_P_1$. Since it is from H, intra-core scheduling is performed as described in more detail below. Reactor2 then proceeds to executing $R_2_P_3$. ❹ Let us return to Reactor1. $R_1_P_1$ handles the I/O from L in the socket queue. Since i i's also from L, this I/O is processed directly as seen in Step 2. ❺ Subsequently, $R_1_P_1$ is executed again, and the I/O request retrieved from the socket

queue is for H. Inter-core scheduling is initially attempted for the I/O, but, at this point, after migration, $Core_1$'s core utilization is lower than U_avg, and $Core_2$'s core utilization is higher than U_avg, so it is not selected as an inter-core scheduling I/O but placed in Reactor1's message queue. ❻ $R_1_P_2$ checks that the I/O from H in the message queue is an intra-core scheduling target I/O and dequeues and enqueues it in the message queue again. Since KVS and NVMe driver processing are not performed for this I/O, $R_1_P_3$ is executed immediately. ❼ Returning to $R_1_P_1$, since the socket queue is empty, it proceeds directly to $R_1_P_2$, where it encounters H's I/O in the message queue once more. It confirms that the execution has been postponed once and performs another round of backwarding (assuming backwarding count is two).

5. Evaluation

In this section, we first describe the experimental setup (Section 5.1). We then analyze the performance of OctoFAS, including I/O latency, throughput, and fairness (Section 5.2).

5.1. Evaluation Setup

Testbed Setup: We implemented OctoFAS using SPDK v.21.10, with two servers connected via a 10 Gbps Ethernet. The client communicated with the servers through NVMe-oF. Both client and server had the same specifications as described in Table 1. The servers configured with six CPU cores operated at a down-clocked speed of 1.80 GHz.

Table 1. Hardware/software specifications.

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

Workloads: We used workloads provided by RocksDB db_bench benchmark [36]. Specifically, "Fill Random" and "Read Random" workloads were used for Put (write) and Get (read), respectively. The key and value sizes were set to 4 B and 16 KB, respectively. Since the KVS used in OctoFAS is based on a simple hash function, the effect of OctoFAS's fair scheduling is independent of the key pattern, key and value sizes, and workload type. The db_bench initiates n threads to concurrently insert or retrieve key-value pairs from OctoFAS. We then classified two clients (flows) based on the number of threads: $flow_1$ (2 threads) and $flow_2$ (8 threads). In each workload, every thread executes 300,000 insert or retrieve operations synchronously. We measured the utilization of individual CPU cores, I/O response time, I/O latency, and throughput. It is important to note that the server measures I/O latency, while I/O response time is measured from the client's perspective.

5.2. Performance and Fairness of Two-Level Fair Scheduling

We analyzed the I/O latency and throughput experienced by each flow, as well as the overall server system throughput and fairness, when applying the detailed algorithms of two-level fair scheduling proposed in OctoFAS. The algorithms include the inter-core scheduling (Section 5.2.1), the intra-core scheduling (Section 5.2.2), and the hybrid scheduling (Section 5.2.3).

5.2.1. Inter-Core Scheduling

The effectiveness of enhancing the overall system throughput and addressing load imbalance issues in SPDK-based KVS through the inter-core scheduling has already been

proved [21,37,38]. However, we recognize that simply transferring I/O requests from all overloaded cores to underloaded cores without considering flow correlation may not be optimal in terms of fairness. To determine what to transfer when performing inter-core scheduling, we measured system throughput, which is defined as the sum of the throughput for each flow, and fairness during the simultaneous execution of $flow_1$ (sparse flow) and $flow_2$ (dense flow). Specifically, we defined cases where no flow is migrated (baseline), only $flow_1$ is migrated, only $flow_2$ is migrated, and both $flow_1$ and $flow_2$ are migrated.

The results are shown in Figure 7a,b. When moving all $flow_i$ to achieve a balanced a load as possible, the overall system throughput increases the most (approximately 4%). However, from a fairness perspective, it is optimal to move only the dense flow ($flow_2$), and moving the sparse flow ($flow_1$) actually degrades fairness. This can be explained by the mechanism of message passing in SPDK, which serves as the medium for load migration in SPDK-based KVS. Through message passing, events migrated to the destination core are processed only after handling all the events in the socket queue that the destination core was processing at that moment. The destination core's reactor thread checks the message queue poller before it executes the P2 poller (see Figure 1). Consequently, when only the requests from the intense (dense) flow are migrated, each execution of migrated events is delayed compared to non-migrated events from sparse flow ($flow_1$). Simultaneously, the non-migrated events from sparse flows on overloaded cores all benefit from the migrations of contenders, resulting in an increase in fairness. On the other hand, when migrating events from sparse flows, the processing of the migrated events is delayed due to the aforementioned waiting time on the message queue. Consequently, events from intense flows are processed first on both overloaded and underloaded cores, leading to performance degradation for sparse flows and eventual fairness degradation.

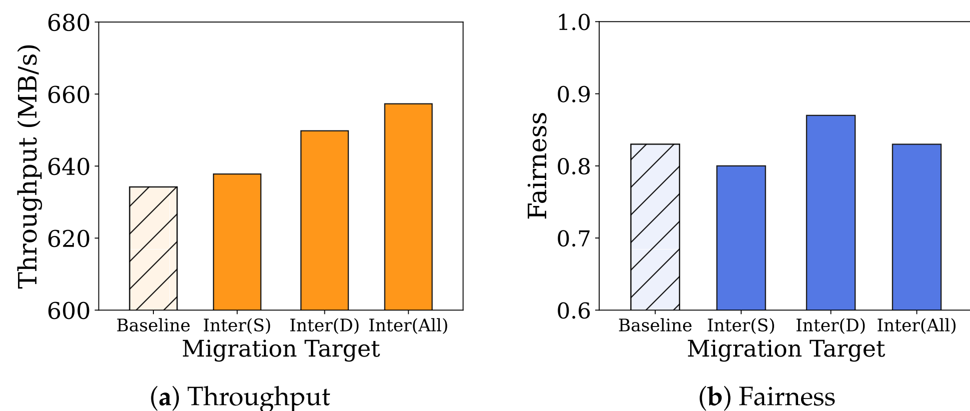


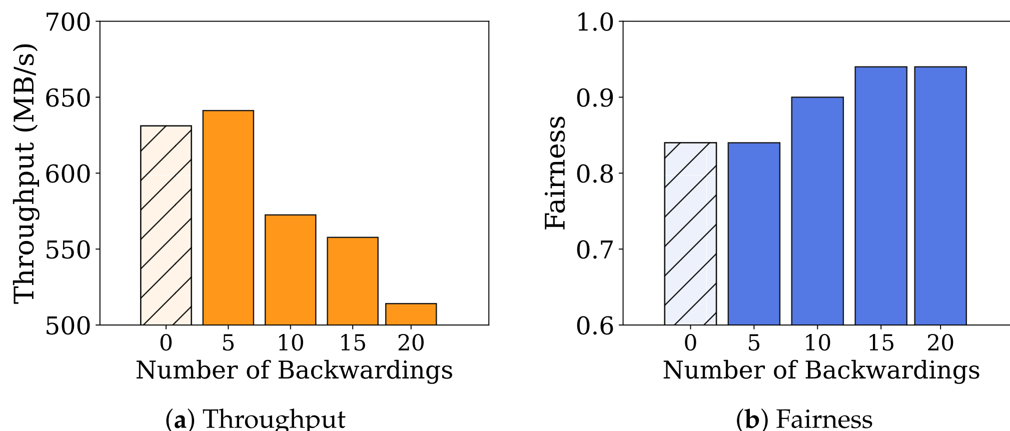
Figure 7. Comparison of the impact of inter-core scheduling on system throughput and fairness based on the migration target. Inter(k) refers to inter-core scheduling that moves the requests of k, where k represents either a sparse tenant or a dense tenant, and All denotes both types of tenants.

5.2.2. Intra-Core Scheduling

In intra-core scheduling, adjusting the delay in handling I/O events for dense flows based on the number of backwarding instances is essential, and understanding its impact on fairness is vital. Therefore, we varied the backwarding counts for the execution of I/O events for dense flows on each core from 0 to 20, increasing by 5 each time. Similar to the previous experiments, we measured the system throughput and fairness.

The results are shown in Figure 8a,b. As the number of backwarding instances in intra-core scheduling increases, fairness improves (approximately 12% increase in case of 20 backwardings), but overall throughput decreases (approximately 19% decrease in case of 20 backwardings). The throughput experiences significant drops starting from the case with 10 backwardings. Note that when the backwarding count increases from zero to five instances, due to the introduction of message passing processing which is the medium for per-queue backwarding, the total throughput temporarily increases. Based on these results,

we found that starting from 10 instances of backwarding, the intra-core scheduling visibly increases fairness, and as the number of instances increases, fairness continues to grow. However, in our environment, we observed that throughput drops without any further increase in fairness starting from 15 instances of backwarding. Therefore, it is evident that setting the number of backwarding instances carefully, considering the trade-off between throughput and fairness in each environment, is necessary.



(**a**) Throughput (**b**) Fairness

Figure 8. Comparison of the impact of intra-core scheduling on system throughput and fairness based on the number of backwardings. A backward value of 0 means a baseline that does not perform intra-core scheduling.

5.2.3. Hybrid Scheduling

Figure 9 shows how the system throughput and fairness change when selecting each scheduling algorithm. Note that both intra-core and hybrid scheduling adopted 20 instances of backwarding, which significantly enhanced fairness and substantially reduced total throughput, as depicted in Figure 8, to assess the best-case fairness and the worst-case total throughput achievable during hybrid scheduling. In addition, for both inter-core and hybrid scheduling, we adopted a migration policy that only migrates events from dense flows. As detailed in Section 5.2.1, this policy stands out as the optimal approach in terms of maximizing fairness while simultaneously boosting overall throughput.

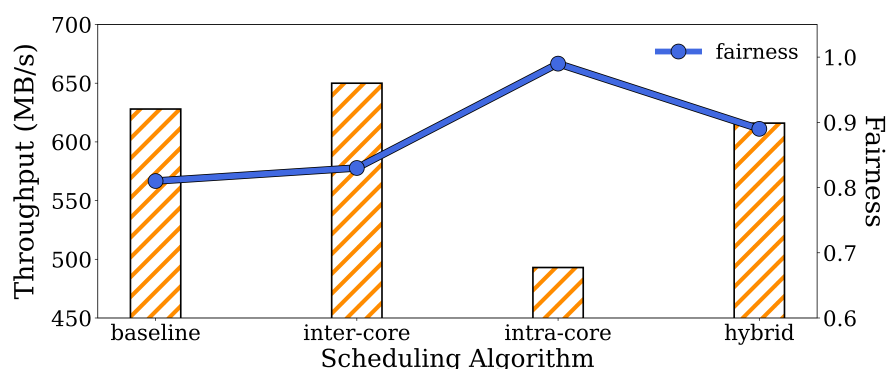


Figure 9. Comparison of the impact of each scheduling algorithm on system throughput and fairness, with migrations for dense flows (inter- and hybrid scheduling) and 20 backwardings performed (intra- and hybrid scheduling).

The results indicate that the hybrid scheduling demonstrated a total system throughput similar to that of the baseline system while achieving an overall improvement in fairness of approximately 10%. When compared to intra-core scheduling, the improvement in fairness against the baseline with hybrid scheduling was relatively small, but it significantly boosted the overall system throughput, improving it approximately 25%. When compared

to inter-core scheduling, hybrid scheduling sacrificed a bit of the overall throughput improvement but achieved an approximately 8% enhanced fairness. As described earlier, we can fine-tune the performance and fairness values by adjusting the migration targets in inter-core scheduling and the number of backwarding instances in intra-core scheduling. To design an SPDK-based KVS that prioritizes both fairness and overall system performance, it is evident that hybrid scheduling should be chosen.

6. Related Work

SPDK for High-Performance Storage Stack: Several studies have adopted SPDK for building high-performance storage systems, including file systems, such as EvFS [15] and FSP [39] and KVS such as SpanDB [14] and TridentKV [16]. However, the focus remained on how to use SPDK, whereas the inherent impact of KVS operations on SPDK performance has not been clearly studied. "Our work focuses on deploying key-value functionality onto the architecture of SPDK-based network clouds and addressing the internal scheduling challenges of SPDK when implementing a KVS within the SPDK framework".

Load Balancing and Fairness Enhancement: Several strategies such as thread or data migration have been explored to balance CPU loads with NUMA architectures [40–43]. These approaches primarily address differences in memory access times incurred by interconnects between processors, e.g., NThread [40] and AASH [42] proposed thread migration as a means of avoiding contention in processor interconnects. There have been efforts directed toward load balancing in storage systems [44–48]. FastResponse [45] proposed several schemes applicable across the Linux storage stack to mitigate I/O interference between co-running low-latency I/O services. The blk-switch [44] achieved low latency and high bandwidth in the block storage layer by adopting a multi-queue design with load balancing and scheduling techniques. OctoFAS mitigates I/O latency spikes by introducing inter-core scheduling, addressing the inherent problems caused by per-core event loop design in SPDK. On the other hand, research on fairness, much like load balancing, has been extensively conducted in various fields, particularly in multi-tenant cloud environments [23–34]. However, there has been a lack of research on fairness within the context of storage solutions structured through SPDK and SPDK-based network storage system. OctoFAS presents a novel dynamic two-level scheduling algorithms that tackles both the load imbalance and unfairness problem inherent in the SPDK-based network key-value storage system.

7. Conclusions

In this study, OctoFAS, our proposed two-level fair scheduler, was found to effectively address the fairness issue in multi-tenant environments using Intel SPDK for network-based key-value storage systems. The two-level fair scheduler comprises inter-core and intra-core scheduling. Through inter-core scheduling, OctoFAS effectively redistributes I/O requests to balance the load across cores, thereby enhancing overall throughput. Additionally, its intra-core scheduling prioritizes requests from less served tenants, ensuring a more equitable distribution of I/O services. Deployed on a Linux cluster with Intel SPDK, OctoFAS not only maintains high system throughput but also improves fairness by 10% compared to the baseline when both scheduling levels operate in a hybrid manner. The hybrid scheduling of OctoFAS demonstrates a significant advancement in achieving both efficiency and fairness in multi-tenant storage systems.

Author Contributions: Writing—original draft, Y.P.; Writing—review & editing, J.P. (Junhyeok Park), J.P. (Junghwan Park) and A.K.; Project administration, Y.K.; Methodology, Resources, K.K.; Conceptualization, Supervision, Writing—review & editing, S.-S.P. All authors have read and agreed to the published version of the manuscript.

Funding: 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), and in part by the National Research Foundation of Korea (NRF) grant funded by the Korean government

(MSIT) (No. NRF-2021R1A2C2014386). And 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.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: Author Kyeongpyo Kim was employed by the company GlueSys Co., Ltd. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

1. Google. LevelDB. Available online: https://github.com/google/leveldb (accessed on 1 November 2023).
2. Meta. RocksDB. Available online: http://rocksdb.org (accessed on 1 November 2023).
3. Bjørling, M.; Gonzalez, J.; Bonnet, P. LightNVM: The linux Open-Channel SSD subsystem. In Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST 17), Santa Clara, CA, USA, 27 February–2 March 2017; pp. 359–374.
4. Samsung. Samsung PM1733/PM1735 NVMe SSD. Available online: https://semiconductor.samsung.com/ssd/enterprise-ssd/pm1733-pm1735/ (accessed on 1 November 2023).
5. Samsung. Samsung NVMe SSD 980 Pro. Available online: https://semiconductor.samsung.com/consumer-storage/internal-ssd/980pro/ (accessed on 1 November 2023).
6. SK hynix. Samsung PM1733/PM1735 NVMe SSD. Available online: https://ssd.skhynix.com/platinum_p41/ (accessed on 1 November 2023).
7. Kim, H.J.; Lee, Y.S.; Kim, J.S. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC 20), New Orleans, LA, USA, 16–20 January 2016; USENIX Association: Berkeley, CA, USA, 2016.
8. Samsung. Samsung Key Value SSD Enables High Performance Scaling. 2017. Available online: https://www.samsung.com/semiconductor/global.semi.static/Samsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf (accessed on 1 November 2023).
9. Lee, C.G.; Kang, H.; Park, D.; Park, S.; Kim, Y.; Noh, J.; Chung, W.; Park, K. iLSM-SSD: An Intelligent LSM-Tree Based Key-Value SSD for Data Analytics. In Proceedings of the 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Rennes, France, 22–25 October 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 384–395.
10. Im, J.; Bae, J.; Chung, C.; Arvind; Lee, S. PinK: High-speed In-storage Key-value Store with Bounded Tails. In Proceedings of the USENIX Annual Technical Conference (ATC), Virtual, 15–17 July 2020; USENIX: Berkeley, CA, USA, 2020; pp. 173–187.
11. Lee, S.; Lee, C.G.; Min, D.; Park, I.; Chung, W.; Sivasubramaniam, A.; Kim, Y. Iterator Interface Extended LSM-tree-based KVSSD for Range Queries. In Proceedings of the 16th ACM International Systems and Storage Conference (SYSTOR'23), Haifa, Israel, 5–7 June 2023.
12. Lim, H.; Han, D.H.; Anderson, D.G.; Kaminsky, M. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In Proceedings of the USENIX Symposium on Networked Systems Design and Implementation, Seattle, WA, USA, 2–4 April 2014.
13. Yang, Z.; Harris, J.R.; Walker, B.; Verkamp, D.; Liu, C.p.; Chang, C.; Cao, G.; Stern, J.; Verma, V.; Paul, L.E. SPDK: A development kit to build high performance storage applications. In Proceedings of the 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Hong Kong, China, 11–14 December 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 154–161.
14. Chen, H.; Ruan, C.; Li, C. SpanDB: A Fast, Cost-Effective LSM-Tree Based KV Store on Hybrid Storage. In Proceedings of the 19th USENIX Conference on File and Storage Technologies, Virtual, 23–25 February 2021; ACM: New York, NY, USA, 2021; pp. 18–32.
15. Yoshimura, T.; Chiba, T.; Horii, H. EvFS: User-level, Event-Driven File System for Non-Volatile Memory. In Proceedings of the 2019 USENIX Annual Technical Conference (USENIX), Renton, WA, USA, 10–12 July 2019; USENIX Association: Berkeley, CA, USA, 2019.
16. Lu, K.; Zhao, N.; Wan, J. *TridentKV: A Read-Optimized LSM-Tree Based KV Store via Adaptive Indexing and Space-Efficient Partitioning*; IEEE: Piscataway, NJ, USA, 2022; pp. 1953–1966.
17. Guz, Z.; Li, H.H.; Shayesteh, A.; Balakrishnan, V. Performance Characterization of NVMe-over-Fabrics Storage Disaggregation. *ACM Trans. Storage* **2018**, *14*, 1–18. [CrossRef]
18. Min, J.; Liu, M.; Chugh, T.; Zhao, C.; Wei, A.; Doh, I.H.; Krishnamurthy, A. Gimbal: Enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM), Virtual, 23–27 August 2021; pp. 106–122.
19. Disaggregated Hyperconverged Storage Will Win in the Enterprise. Available online: https://www.nextplatform.com/2017/12/04/disaggregated-hyperconverged-storage-will-win-enterprise/ (accessed on 1 November 2023).
20. Klimovic, A.; Kozyrakis, C.; Thereska, E.; John, B.; Kumar, S. Flash storage disaggregation. In Proceedings of the Eleventh European Conference on Computer Systems, London, UK, 18–21 April 2016. [CrossRef]
21. Park, Y.; Park, J.; Awais, K.; Park, J.; Lee, C.G.; Chung, W.; Kim, Y. OctoKV: An Agile Network-Based Key-Value Storage System with Robust Load Orchestration. In Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Stony Brook, NY, USA, 16–18 October 2023; pp. 145–152.

22. Nesbit, K.J.; Aggarwal, N.; Laudon, J.; Smith, J.E. Fair Queuing Memory Systems. In Proceedings of the 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Orlando, FL, USA, 9–13 December 2006.
23. Mutlu, O.; Moscibroda, T. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Chicago, IL, USA, 1–5 December 2007.
24. Valente, P.; Checconi, F. High Throughput Disk Scheduling with Fair Bandwidth Distribution. *IEEE Trans. Comput.* **2010**, *59*, 1172–1186. [CrossRef]
25. Valente, P.; Checconi, F. Fairness Metrics for Multi-Threaded Processors. *IEEE Comput. Archit. Lett.* **2011**, *10*, 4–7.
26. Ebrahimi, E.; Lee, C.J.; Mutlu, O.; Patt, Y.N. Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. *ACM Trans. Comput. Syst.* **2012**, *30*, 1–35. [CrossRef]
27. Park, S.; Shen, K. FIOS: A Fair, Efficient Flash I/O Scheduler. In Proceedings of the the 10th USENIX Conference on File and Storage Technologies (FAST), San Jose, CA, USA, 15–17 February 2012.
28. Shue, D.; Freedman, M.J.; Shaikh, A. Performance isolation and fairness for multi-tenant cloud storage. In Proceedings of the the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI), Hollywood, CA, USA, 8–10 October 2012; pp. 349–362.
29. Shen, K.; Park, S. FlashFQ: A fair queueing I/O scheduler for flash-based SSDs. In Proceedings of the 2013 USENIX conference on Annual Technical Conference (ATC), San Jose, CA, USA, 26–28 June 2013; pp. 67–78.
30. Subramanian, L.; Seshadri, V.; Kim, Y.; Jaiyen, B.; Mutlu, O. MISE: Providing performance predictability and improving fairness in shared main memory systems. In Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), Shenzhen, China, 23–27 February 2013.
31. Subramanian, L.; Lee, D.; Seshadri, V.; Rastogi, H.; Mutlu, O. BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling. *IEEE Trans. Parallel Distrib. Syst.* **2016**, *27*, 3071–3087. [CrossRef]
32. Subramanian, L.; Lee, D.; Seshadri, V.; Rastogi, H.; Mutlu, O. The Blacklisting Memory Scheduler: Achieving high performance and fairness at low cost. In Proceedings of the 2014 IEEE 32nd International Conference on Computer Design (ICCD), Seoul, Republic of Korea, 19–22 October 2014.
33. Tavakkol, A.; Sadrosadati, M.; Ghose, S.; Kim, J.; Luo, Y.; Wang, Y.; Ghiasi, N.M.; Orosa, L.; Gómez-Luna, J.; Mutlu, O. FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives. In Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), Los Angeles, CA, USA, 1–6 June 2018.
34. Liu, R.; Tan, Z.; Shen, Y.; Long, L.; Liu, D. Fair-ZNS: Enhancing Fairness in ZNS SSDs through Self-balancing I/O Scheduling. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2022**. [CrossRef]
35. Intel. Message Passing and Concurrency. 2019. Available online: https://spdk.io/doc/concurrency.html (accessed on 1 November 2023).
36. Meta. db_bench. Available online: https://github.com/facebook/rocksdb/wiki/Benchmarking-tools (accessed on 1 November 2023).
37. Afzal, S.; Kavitha, G. Load balancing in cloud computing—A hierarchical taxonomical classification. *J. Cloud Comput.* **2019**, *8*, 22. [CrossRef]
38. Shafiq, D.A.; Jhanjhi, N.Z.; Abdullah, A. Load balancing techniques in cloud computing environment: A review. *J. King Saud Univ.-Comput. Inf. Sci.* **2022**, *34*, 3910–3933. [CrossRef]
39. Liu, J.; Arpaci-Dusseau, A.C.; Arpaci-Dusseau, R.H.; Kannan, S. File Systems as Processes. In Proceedings of the 2019 USENIX Annual Technical Conference (USENIX), Renton, WA, USA, 10–12 July 2019; USENIX Association: Berkeley, CA, USA, 2019.
40. Wang, Y.; Jiang, D.; Xiong, J. NUMA-Aware Thread Migration for High Performance NVMM File Systems. In Proceedings of the 36th International Conference on Massive Storage Systems and Technology (MSST), Santa Clara, CA, USA, 29–30 October 2020.
41. Dashti, M.; Fedorova, A.; Funston, J.; Gaud, F.; Lachaize, R.; Lepers, B.; Quéma, V.; Roth, M. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Houston, TX, USA, 16–20 March 2013; pp. 381–394.
42. Kazempour, V.; Kamali, A.; Fedorova, A. AASH: An Asymmetry-Aware Scheduler for Hypervisors. *ACM SIGPLAN Not.* **2010**, *45*, 85–96. [CrossRef]
43. Blagodurov, S.; Fedorova, A.; Zhuravlev, S.; Kamali, A. A Case for NUMA-Aware Contention Management on Multicore Systems. In Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), Vienna, Austria, 11–15 September 2010; pp. 55–64.
44. Hwang, J.; Vuppalapati, M.; Peter, S.; Agarwal, R. Rearchitecting Linux Storage Stack for μs Latency and High Throughput. In Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21), Virtual, 14–16 July 2021; pp. 113–128.
45. Liu, M.; Liu, H.; Ye, C.; Liao, X.; Jin, H.; Zhang, Y.; Zheng, R.; Hu, L. Towards low-latency I/O services for mixed workloads using ultra-low latency SSDs. In Proceedings of the 36th ACM International Conference on Supercomputing, Virtual, 28–30 June 2022; pp. 1–12.
46. Ma, L.; Liu, Z.; Xiong, J.; Jiang, D. QWin: Core Allocation for Enforcing Differentiated Tail Latency SLOs at Shared Storage Backend. In Proceedings of the 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS), Bologna, Italy, 10–13 July 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 1098–1109.

47. Hahn, S.S.; Lee, S.; Yee, I.; Ryu, D.; Kim, J. Fasttrack: Foreground app-aware i/o management for improving user experience of android smartphones. In Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC 18), Boston, MA, USA, 11–13 July 2018; pp. 15–28.
48. Zhang, J.; Kwon, M.; Gouk, D.; Koh, S.; Lee, C.; Alian, M.; Chun, M.; Kandemir, M.T.; Kim, N.S.; Kim, J.; et al. FlashShare: Punching through server storage stack from kernel to firmware for ultra-low latency SSDs. In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), Carlsbad, CA, USA, 8–10 October 2018; pp. 477–492.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.