i-NVMe: Isolated NVMe over TCP for a Containerized Environment

Seongho Lee, Ikjun Yeom and Younghoon Kim*
Department of Computer Science and Engineering, College of Computing and Informatics
Sungkyunkwan University, Suwon, Republic of Korea
Emails: {sk1528, ikjun and yhoon}@skku.edu

IEEE INFOCOM 2023 - IEEE Conference on Computer Communications | 979-8-3503-3414-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/INFOCOM53939.2023.10228889

***Abstract*—Non-Volatile Memory Express (NVMe) over TCP is an efficient technology for accessing remote Solid State Drives (SSDs); however, it may cause a serious interference issue when used in a containerized environment. In this study, we propose a CPU isolation scheme for NVMe over TCP in such an environment. The proposed scheme measures the CPU usage of the NVMe over TCP worker, charges it to containers in proportion to their NVMe traffic, and schedules containers to ensure isolated sharing of the CPU. However, because the worker runs with a higher priority than normal containers, it may not be possible to achieve CPU isolation with container scheduling alone. To solve this problem, we also control the CPU usage of the worker by throttling NVMe over TCP traffic. The proposed scheme is implemented on a real testbed for evaluation. We perform extensive experiments with various workloads and demonstrate that the scheme can provide CPU isolation even in the presence of excessive NVMe traffic.**

***Index Terms*—File and storage system, Scheduling, Resource management**

I. Introduction

Recently, disaggregated storage has been widely deployed in modern data center to satisfy the increasing demand for huge data storage volume of ML and big data processing [1], [2]. Unlike traditional storage systems, such as direct-attached storage and storage area networks (SAN), in which capacity expansion is limited by physical drive bays and expansion chassis, disaggregated storage is a flexible and scalable solution since a number of storage devices can be allocated to any server using a high-speed network fabric.

Non-Volatile Memory Express over Fabrics (NVMe-oF) is an enabling technology to realize disaggregated storage. In NVMe-oF, storage devices (targets), more precisely NVMe SSDs, are attached to a high-speed network infrastructure, and servers (hosts) can access them via the network infrastructure. To provide transparent access to NVMe SSDs, NVMe-oF host and target drivers encapsulate NVMe commands and data within a network protocol, and servers can access them like local NVMe SSDs.

Currently, two network protocols are supported by NVMe-oF, remote direct memory access (RDMA) and transmission control protocol (TCP). RDMA was originally developed for InfiniBand as a proprietary protocol [3], [4]. It is now opened to the public and available for Ethernet as well [5]. In RDMA, packet processing is offloaded to the NIC hardware, and it is possible to provide ultra low latency and high bandwidth. NVMe over RDMA inherits the advantages of RDMA, and hosts can access the target SSDs with very low latency. However, it also inherits the limitations of RDMA, requiring specialized NICs and switches, and vulnerability of congestion, and this makes it difficult to deploy in a large scale network. Compared to NVMe over RDMA, the advantages and disadvantages of NVMe over TCP are clear such that: (a) TCP is the most popular transport layer protocol and robust to congestion; (b) NVMe over TCP can be easily deployed for a large scale network without specialized equipment; but (c) In performance wise, network latency is higher than that of NVMe over RDMA because protocol complexity is high to handle congestion, and packet processing mostly relies on software stacks. Recent studies on NVMe over TCP have attempted to optimize and/or offload protocol stacks to reduce network latency, and it seems that their efforts make significant progress.

To deploy NVMe over TCP, besides the network performance, a CPU usage issue also needs to be resolved. Compared when accessing local storage, accessing remote NVMe SSDs over TCP consumes significant additional CPU resources for processing NVMe and TCP protocols. On the target side, it can be minimized with specialized hardware for offloading NVMe and TCP packet processing [6], [7]. On the host side running on a commodity server, however, substantial works for processing NVMe over TCP protocol remain on the software stack even if we consider recent NICs with TCP offloading engine (TOE). This additional CPU consumption results in performance degradation of applications, and it can cause more serious problems when a server co-hosts CPU-intensive containers and data-intensive containers. The CPU usage for processing heavy I/O traffic from data-intensive containers can increase the completion time of CPU-intensive containers significantly, and this motivates our work in this paper to provide CPU isolation for NVMe over TCP in a containerized environment.

Providing CPU isolation for NVMe over TCP is not trivial because NVMe and TCP packets are processed by kernel worker threads on behalf of the corresponding containers. Several isolation schemes have been proposed in a containerized environment for various resources such as CPU [8], [9], network bandwidth [10], [11], and block I/O [12], but they are not directly applicable for NVMe over TCP since they

*Younghoon Kim is the corresponding author.

focus on isolation among containers, and resources consumed for specific containers by kernel worker threads are not considered. Recently, a CPU isolation scheme, called Iron, has been proposed in [13]. Iron measures the CPU usages for processing network packets and charges them to the corresponding containers so that other containers can be protected from containers generating heavy network traffic. However, Iron also cannot guarantee CPU isolation for NVMe over TCP because packets considered in Iron are issued directly from containers whereas NVMe over TCP packets are issued by NVMe over TCP kernel workers on behalf of actual containers.

In this paper, we propose a scheme, called isolated NVMe over TCP (i-NVMe), to provide CPU isolation in a containerized environment. In i-NVMe, CPU usages for processing NVMe and TCP packets are carefully measured and charged to the corresponding containers. However, accurate measuring and charging are not enough to provide CPU isolation since NVMe kernel worker threads are scheduled with a high priority. To enforce CPU isolation including the NVMe over TCP kernel worker thread, we provide a dynamic throttling scheme to control NVMe over TCP traffic. To implement i-NVMe, we modify the Linux Completely Fair Scheduling (CFS) scheduler logic and NVMe over TCP driver logic. It is demonstrated that i-NVMe can protect CPU-intensive containers from data-intensive containers while maintaining the efficiency of NVMe over TCP. This paper makes the following significant contributions as follows,

- It is addressed that the CPU usages consumed by NVMe over TCP worker threads can cause significant degradation in the performance of CPU-intensive containers. Our measurements reveal that the completion time of CPU-intensive workload can increase up to almost 2 times. To the best of our knowledge, this is the first attempt to investigate the impact of NVMe over TCP in a containerized environment.
- A novel scheme, i-NVMe, is proposed to limit the impact of NVMe over TCP on the performance of CPU-intensive containers. i-NVMe measures the CPU usages of NVMe over TCP worker threads, charges them to each container in proportion to its I/O usage, and dynamically throttles them to enforce isolation of CPU resources. Additional tail latency caused by throttling is also handled in a k-split manner.
- We evaluate i-NVMe through extensive experiments on a real testbed. It is shown that i-NVMe can provide accurate CPU isolation with both benchmark workloads and real distributed computing workloads.

The remaining part of this paper is organized as follows: In Section II, we present background and motivation of i-NVMe; In Section III, we present the design and implementation of i-NVMe; In Section IV, we evaluate the performance of i-NVMe with extensive experiments; In Section V, related work is presented; and In Section VI, we conclude this paper.

II. Background and Motivation

In this section, we describe how NVMe over TCP works in the Linux-containerized environment and how these processing steps interfere with CPU resources allocated to other containers.

A. NVMe over TCP Host Driver

In NVMe over TCP, there are two types of nodes: host nodes and target nodes. Host nodes access remote SSDs in a target node using the NVMe over TCP host driver, and the target node handles received NVMe over TCP requests from hosts. In this work, we focus on the host-side driver, which runs on the containerized environment. The NVMe over TCP host driver is implemented using Linux's block layer structure and a TCP network stack without modification. More specifically, the NVMe over TCP host driver receives requests from the block layer of Linux, generates NVMe over TCP commands and protocol data units (PDUs), and sends them to the target node through the TCP network stack. To convert the block layer's requests into NVMe over TCP commands and forward them to the TCP network stack, the current NVMe over TCP host driver uses the `workqueue` in Linux [14]. When a host node is connected to a target node with NVMe over TCP, the host driver opens one TCP connection per core and a worker thread, called *nvme_tcp_wq*, with WQ_HIGHPRI option. An *nvme_tcp_wq* thread is scheduled when a user requests block I/Os to a remote SSD. It encapsulates requests and sends them to the target node over the TCP connection. The transmitted requests are processed by the NVMe over TCP target driver and the NVMe SSD of the target node, and the responses will be sent over the same TCP connection. The *nvme_tcp_wq* thread then handles those responses and returns the results to the upper layer.

The detailed operation of a *nvme_tcp_wq* thread is as follow: When it is scheduled, it sends requests in its list and handles responses from the TCP connection as many as possible until either the deadline is reached or there is no more request or response. Here, the deadline is set by the NVMe over TCP host driver to prevent the *nvme_tcp_wq* thread from consuming too much CPU resources, and the default value of the deadline is 1 ms. It is re-scheduled either (a) immediately if it was scheduled out due to the deadline, or (b) upon arrival of a new request if there was no more request. The operation of a *nvme_tcp_wq* thread is effective to handle NVMe I/O requests with a minimal latency, but there is a risk of overusing the CPU. In our experiments, we can find that almost 50% of CPU resources can be consumed by the *nvme_tcp_wq* thread when IOPS is high.

B. Linux CFS Scheduler

We present a brief background on Linux scheduling in a containerized environment to understand how NVMe over TCP can break CPU isolation. Most tasks, including user threads and a *nvme_tcp_wq* thread running on a containerized environment, are scheduled using the CFS scheduler in Linux [9].

The CFS scheduler operates to ensure that each task can consume as much CPU as the time distributed in proportion to its `nice` value in Linux, which indicates each task's priority.

For instance, when the *nvme_tcp_wq* thread, assigned the highest nice value using the WQ_HIGHPRI option, and a user thread are running on the same core, the user thread is scheduled only for a small portion of the CPU time. Since *nvme_tcp_wq* is scheduled out when there is no I/O to handle or its deadline is reached, the actual usage does not appear extreme, but, when heavy I/O occurs, *nvme_tcp_wq* occupies the CPU first and is executed.

The CPU usages of containers are limited by cgroups. The CFS scheduler is implemented such that each container can use as much CPU as a `quota` during a `period`, and the user can adjust the usage by setting the quota for each container. As explained in [9], [13], the CFS scheduler keeps a variable, `runtime`, which indicates the CPU usage of a container within the current period. When the runtime exceeds its quota, the container is throttled and rescheduled in the next period. A container's runtime is reset for each period.

C. CPU Isolation Violation of NVMe over TCP

When we use NVMe over TCP to access remote SSDs, there are two possible risks to violate CPU isolation. First, the CPU usage of a *nvme_tcp_wq* thread increases as the NVMe traffic increases. Since the kernel thread is running with the highest priority, it can account for CPU time as much as it needs. Then, containers cannot use their quotas over a period, and isolation breaks.

Second, as described in [13], sending and receiving TCP packets require frequent CPU preemption for interrupt handling. In Linux, packets are processed by two parts of interrupts, a top-half (hardware interrupt) and a bottom-half (software interrupt). To limit the impact of hardware interrupts, the top-half is only used to interface with NICs and to schedule the bottom-half. Actual processing for packets is performed by the bottom-half. Since software interrupts run in process context, CPU quota of any container can be used to process packets for other containers, and this also breaks isolation.

To demonstrate how NVMe over TCP harms the performance of other containers, we perform preliminary experiments with the following experimental environment: n containers are created to share one core, and each container's CPU quota is configured as $period/n$ so that all containers can equally occupy the CPU. We use a 40 Gbps TCP/IP network to connect an NVMe over TCP host and a target with NVMe SSDs. A detailed explanation of the experimental environment is provided in Section IV. Among n containers, one (denoted as `victim`) runs a CPU-intensive `sysbench` workload, and the others (denoted as `interferers`) run both FIO [15] to generate NVMe over TCP traffic and `sysbench` to consume the remaining quota. Because these containers share the same core, NVMe over TCP interference occurs. Note that we use 4 KB mixed workloads that issue both 4 KB random read and 4 KB random write requests. We vary NVMe traffic IOPS, and measure the victim's completion time to observe the impact of NVMe over TCP on CPU intensive workload. Then, we calculate a *degrading factor*, the fraction of the victim's completion time when *nvme_tcp_wq* is scheduled versus when only sysbench consumes the CPU.

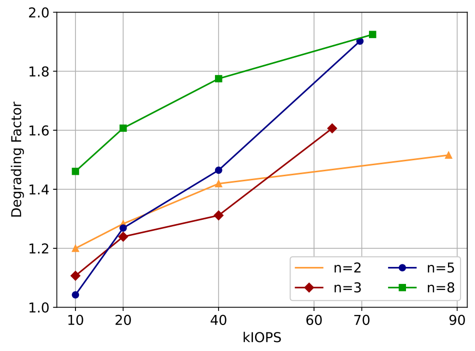


Fig. 1: Degrading Factor with interferers

Figure 1 shows the results with various numbers of interferers, and the interferer's target IOPS varies from 10 K, 20 K, 40 K, to unlimited. The x-axis denotes the sum of IOPS of interferers, and the y-axis indicates the degrading factor. As IOPS increases, the CPU usage required to handle I/O requests also increases, which leads to a higher degrading factor with more severe interference. It is also shown that the degrading factor increases as more interferers are in presence. In these experiments, increase in NVMe I/O usage can increase the degrading factor to 1.92. This means that interferers will consume more CPU than their quota through *nvme_tcp_wq*, and this additional uncharged CPU usage can break CPU isolation in a containerized environment.

Our observations and their implications are summarized as follows:

- The CPU usage consumed by *nvme_tcp_wq*, a worker thread for NVMe over TCP, should be taken into consideration when calculating containers' CPU usages.
- Network traffic from the NVMe over TCP driver is handled by interrupts, and it consumes the quota of other containers by preempting them. The sources of traffic should be specified for precise CPU usage accounting.
- *nvme_tcp_wq* is scheduled with the highest priority in the Linux CPU scheduler, so it needs to be controlled to reduce excessive CPU usage that lowers other containers' performance.

To this end, we conclude that it is necessary to consider above observations when designing a new NVMe over TCP host driver with CPU isolation.

III. Design and Implementation of i-NVMe

To realize CPU isolation in a containerized environment, we design i-NVMe scheme performing the followings: The CPU usage of the worker for handling NVMe over TCP traffic is firstly measured. This CPU usage is considered as a part of interferers' operations, and it is charged to the corresponding interferers. By charging, we can enforce interferers to consume less CPU usages, but still the worker can break CPU isolation because it runs with the highest priority. To solve this problem, i-NVMe throttles the worker with a dynamically adjusted

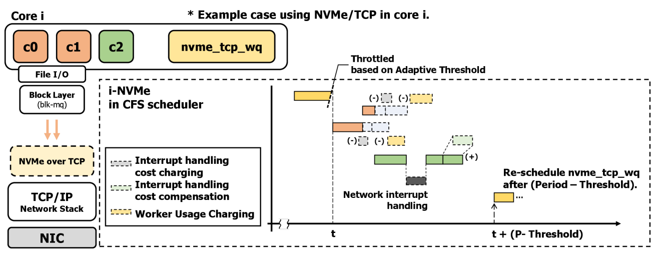

Core i
* Example case using NVMe/TCP in core i.
c0
c1
c2
nvme_tcp_wq
File I/O
Block Layer
(blk-mq)
NVMe over TCP
TCP/IP
Network Stack
NIC
i-NVMe
in CFS scheduler
Interrupt handling cost charging
Interrupt handling cost compensation
Worker Usage Charging
Throttled based on Adaptive Threshold
(-)
(-)
(-)
(-)
(+)
Network interrupt handling
Re-schedule nvme_tcp_wq after (Period – Threshold).
...
t
t + (P- Threshold)

Fig. 2: The structure of i-NVMe

threshold when isolation violation is detected. When throttling the worker, to avoid unnecessary latency of requests arriving during the throttling time, we split the throttling time into several short pieces (k-split throttling). The basic operations of i-NVMe are summarized as follows,

1. Measuring the CPU usages consumed by the NVMe over TCP host driver and TCP network stack.
2. Charging measured usages in the Linux CFS scheduler.
3. Throttling NVMe over TCP worker thread in k-split manner to prevent it from overusing CPU too much.

Figure 2 shows a brief structure of i-NVMe, including an example of NVMe over TCP applied to a containerized environment. In this example, c0 and c1 are interferers, and c2 is the victim. When two containers c0 and c1 request I/O in their applications, *nvme_tcp_wq* is scheduled to handle them. i-NVMe measures *nvme_tcp_wq* worker usages and the preempted time for interrupt handling for *nvme_tcp_wq* traffic. The CPU usages of the worker and the preempted time are charged to the interferers in proportion to their NVMe traffic, and also the preempted time from the victim is added to the victim's quota for compensation. Meanwhile, NVMe packet processing is properly throttled to meet the CPU isolation conditions.

A. CPU Usage Measurements

On the NVMe over TCP driver, `nvme_tcp_try_send` and `nvme_tcp_try_recv` functions are invoked in *nvme_tcp_wq* to handle requests and responses, respectively. To measure the per container *nvme_tcp_wq* usage, we account for the elapsed time of each function call. Then, we retrieve the owner of each handled request using the block layer request structure, and update the *nvme_tcp_wq* usage to each container's `task_group` structure in Linux.

The algorithm described in Iron [13] is used for measuring the interrupt handling cost that occurs when transmitting or receiving each packet. In Iron, the elapsed times of `netif_receive_skb` and `net_tx_action` functions are calculated when transmitting and receiving packets. At the same time, they also measure fixed costs such as hardIRQ handling, softIRQ handling, and socket buffer (skb) garbage collection, and distribute them to each container in a weighted fashion. Similarly, we implement the logic to measure the network traffic-handling cost from the Linux network stack and check whether the processed packets are owned by *nvme_tcp_wq*. If a packet is sent from or destined to the *nvme_tcp_wq*, we accumulate its measured cost in the *nvme_tcp_wq* context. Then, we distribute the measured costs from the network stack in proportion to each container's I/O usage. This weighted fashion distribution is essential for i-NVMe because it is impossible to distinguish whose requests are on the network stack because the actual packets are owned by *nvme_tcp_wq*. At the same time, we update the `net_gain`, which indicates the preempted time of each `task_group`. We need this two-step distribution method because it is not possible to identify which container should be charged the calculated costs during network stack processing.

Algorithm 1 Global runtime update

```
 1: if $runtime > 0$ then
 2:     $need\_reset \leftarrow True$
 3: end if
 4: $wq\_usage \leftarrow wq\_usage\_of(task\_group)$
 5: if $wq\_usage > 0$ then
 6:     if $runtime > wq\_usage$ then
 7:         $runtime \leftarrow runtime - wq\_usage$
 8:         $set\_wq\_usage(task\_group, 0)$
 9:     else
10:         $runtime \leftarrow 0$
11:         $remain\_usage \leftarrow wq\_usage - runtime$
12:         $set\_wq\_usage(task\_group, remain\_usage)$
13:     end if
14: else
15:     if $need\_reset$ is $True$ then
16:         $set\_reset\_flag(True)$
17:     end if
18:     $net\_gain \leftarrow net\_gain\_of(task\_group)$
19:     $runtime \leftarrow runtime + net\_gain$
20: end if
21: if $group\_idled()$ and $runtime > 0$ then
22:     $runtime \leftarrow 0$
23: end if
24: $set\_timer(now + period)$
```

B. Integrating with the CFS Scheduler

Limiting inteferers' runtime based on uncharged CPU usage measured in Section III-A is implemented on the Linux CFS scheduler. In the Linux CFS scheduler, each container is managed as a `task_group`, and it is allowed to run for a given `quota` during a `period`. So in short, we can limit each container's CPU usages by subtracting un-imposed CPU usages measured above from its `quota`. To limit the total CPU usage (by applications and by NVMe traffic) of a container within its quota, we subtract the measured CPU usage for NVMe traffic from the quota for each container. In this section, we describe the charging scheme of i-NVMe in the Linux CFS scheduler. In particular, we present two algorithms for charging uncharged CPU usages caused by remote I/Os to the global and local state updates of the Linux CFS scheduler.

The CFS scheduler refills the quota at the runtime of each `task_group` for each period in the global state. As a local state, the `rt_remain` value is assigned as much as `slice` from runtime. As each `task_group` consumes CPU, the CFS scheduler decrements `rt_remain` of the

Algorithm 2 Local state update

```
1: $amount \leftarrow 0$
2: $min\_amount \leftarrow slice - rt\_remain$
3: $wq\_usage \leftarrow wq\_usage\_of(task\_group)$
4: if $wq\_usage > 0$ then
5:     if $runtime > wq\_usage$ then
6:         $runtime \leftarrow runtime - wq\_usage$
7:         $set\_wq\_usage(task\_group, 0)$
8:     else
9:         $runtime \leftarrow 0$
10:        $remain\_usage \leftarrow wq\_usage - runtime$
11:        $set\_wq\_usage(task\_group, remain\_usage)$
12:    end if
13: end if
14: if $runtime > 0$ then
15:     $amount \leftarrow min(runtime, min\_amount)$
16:     $runtime \leftarrow runtime - amount$
17: end if
18: $rt\_remain \leftarrow rt\_remain + amount$
```

Algorithm 3 nvme_tcp_wq throttling

```
1: if $throttled$ then
2:     return NULL
3: end if
4: if $runtime \geq \frac{T}{k}$ then
5:     $num\_throttle \leftarrow num\_throttle + 1$
6:     if $num\_throttle$ is $k$ then
7:         $T \leftarrow T + \alpha\mu s$
8:         $num\_throttle \leftarrow 0$
9:     end if
10:    $throttled \leftarrow True$
11:    $\tau \leftarrow \frac{Period - T}{k}$
12:    $schedule\_work\_at(nvme\_tcp\_wq, \tau)$
13:    return NULL
14: else
15:    $request \leftarrow get\_first\_request\_from\_queue()$
16:    return request
17: end if
```

task_group. When rt_remain is less than zero, the CFS scheduler attempts to refill rt_remain from the global runtime variable. When both rt_remain and runtime hit zero, the task_group is throttled. We charge the measured *nvme_tcp_wq* usage to interferers, as described in Algorithm 1 and 2.

Algorithm 1 represents the global runtime update scheme. In i-NVMe, wq_usage includes the CPU usage consumed by *nvme_tcp_wq* and the distributed network interrupts handling costs, as explained in Section III-A. Therefore, wq_usage is subtracted from the runtime when $wq\_usage > 0$. If wq_usage is larger than its remaining runtime, the runtime is reset to 0, and the remaining wq_usage is updated to the task_group's wq_usage again as in Lines 10-12. When wq_usage is 0, it means that this task_group does not use I/O, and net_gain, which is the preempted time to handle the network traffic of *nvme_tcp_wq*, is added to runtime on Line 19 in Algorithm 1.

Similarly, Algorithm 2 shows the local rt_remain update scheme. When $rt\_remain \leq 0$, the CFS scheduler calls this function and refills rt_remain using the runtime value. Before the value amount is determined, i-NVMe decrements the runtime as wq_usage when $wq\_usage > 0$.

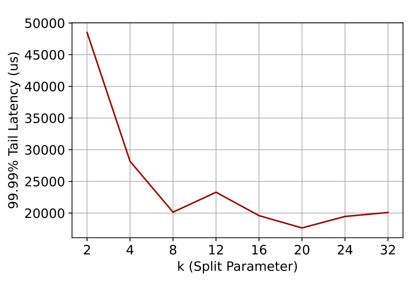
C. NVMe over TCP Worker Throttling

To observe the impact of the aforementioned measuring and charging scheme, we employ the scheme on the same experiment setup described in Section II-C. Although the measuring and charging scheme is enabled, and it works correctly, the degrading factor of the victim is still measured high at 1.411 while it was 1.569 without the scheme.

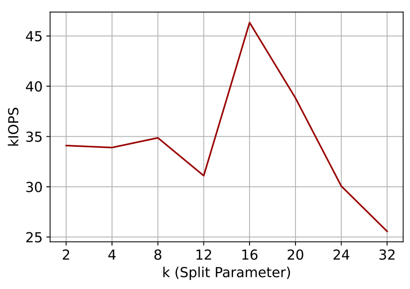
This occurs for two reasons: (a) Limiting the CPU usages of interferers is not sufficient to increase the CPU usage of the victim. Since the CFS scheduler selects the process (the scheduling unit) with the lowest vruntime, and *nvme_tcp_wq* has the highest priority and the lowest vruntime, it is scheduled first. This means that *nvme_tcp_wq* may consume the remaining CPU time due to the CPU quota reduction of interferers; and (b) This occurs because *nvme_tcp_wq* is frequently scheduled when I/O requests are generated at a high rate. *nvme_tcp_wq* is scheduled when a request is enqueued, and the previous *nvme_tcp_wq* ends owing to its deadline. Therefore, when many I/O requests occur, *nvme_tcp_wq* is also frequently scheduled, which leads to high CPU consumption of *nvme_tcp_wq*.

To cope with this issue and make the victim container consume as much CPU as its quota, we throttle *nvme_tcp_wq* when its CPU usage exceeds a threshold T. It is initially set as a half of the total quota of interferers, ($io\_used\_quota/2$). Algorithm 3 presents the throttling scheme. On nvme_tcp_fetch_request, when *nvme_tcp_wq*'s runtime exceeds T, the worker does not fetch the next request from the queue until a new request arrives from the process. Meanwhile, to prevent *nvme_tcp_wq* from being throttled too long, we schedule *nvme_tcp_wq* after τ. Line 12 schedules *nvme_tcp_wq* after τ. We set τ as (period$-T$) so that *nvme_tcp_wq* can consume CPU only as much as T during a period.

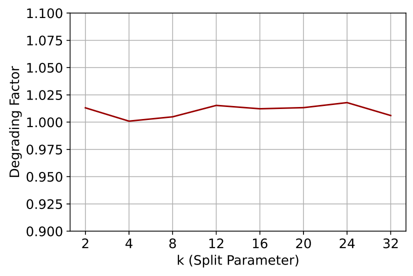
The biggest concern with the throttling scheme is that it may increase the latency of each I/O operation. In particular, the tail latency can increase because I/O operations cannot be handled when the worker is throttled. We run the same experiment in Section II-C with $n = 2$ and measure the average and 99.99% tail latency of a 4KB random read and 4KB random write request of FIO application. The average latency is about $151.57\mu s$ and $108.53\mu s$ for random read and random write, respectively. Most requests can be handled normally, but some requests can be delayed by τ due to throttling. These delays lead to significant increase in the request's tail latency, and the 99.99% tail latency is over 58 ms, while it is measured as 13 ms with the default NVMe over TCP driver. To reduce the I/O tail latency, we design a k-split throttling scheme, in which we divide a CPU time (T) and a throttling time (τ) into k small pieces, and alternate them. Lines 4-13 in Algorithm 3 describe how to split the total throttle time into k small sub-throttles. The impact of the k-splits throttling scheme will be



(a) 99.99% Tail latency



(b) kIOPS



(c) Degrading Factor of victim

Fig. 3: Performance of i-NVMe when k varies from 2 to 32

discussed in Section III-E.

The remaining challenge is how to find a proper threshold T. We should carefully choose T since i-NVMe isolates performance of containers based on T. However, finding an optimal T is not trivial because it is dependent on the numbers of CPU-intensive and data-intensive containers, the amount of NVMe traffic, and network conditions, and they are dynamically changed over time.

To overcome this issue, we design an adaptive threshold method. In this method, we increase the threshold T with α μs when *nvme_tcp_wq* is throttled as long as victim containers fully consume their CPU quotas. This allows *nvme_tcp_wq* to consume more CPU usage for processing I/O requests and responses for data-intensive containers. When victims fail to exhaust their CPU quotas at the end of each period, we detect CPU isolation violation, and decrease T to $max\{T - 0.5(T - minT), minT\}$ in *nvme_tcp_wq* routine, where `minT` is $io\_used\_quota/2$. Here, the max operation is used to avoid too small T. Notes that, only one *nvme_tcp_wq* per a core operates in a single flow independent of the I/O behavior of each container, so it can operate without any locking methods.

D. Adaptive Threshold Method Parameters

In the adaptive threshold method, α and $minT$ primarily determine how much IOPS i-NVMe can serve by increasing the CPU usage that *nvme_tcp_wq* can consume. As α increases, the average CPU usage of *nvme_tcp_wq* also increases, and accordingly, the average IOPS also increases. At the same time, this can worsen CPU interference. To find the parameter pairs that properly maintain both i-NVMe's IOPS and CPU isolation, we perform several empirical observations. We vary α from 1,000 μs to 20,000 μs while performing the experiment in Section II-C with $n = 2$. Detailed experimental results are omitted due to space limitation, but it is noted that both IOPS and degrading factor slightly and linearly increase as we increase α.

Based on the experiment results and the observation, we suggest 5,000 to 8,000 μs for the range of α so that i-NVMe can maintain both NVMe I/O performance and CPU isolation. Within this range, the user can make i-NVMe work I/O friendly or conservatively preserve CPU isolation, depending on the workload type. It is important to note that α values in the suggested range will properly work with varied network conditions.

$minT$ determines the minimum CPU usage that *nvme_tcp_wq* may be guaranteed. $io\_used\_quota/2$ in Section III-C is used to guarantee that *nvme_tcp_wq* consumes half of interferers' allowed CPU usage. If containers (running both computations and remote I/Os) require more CPU usages for computation, $minT$ can be configured to a lower value.

E. k-Split Throttling Parameter

Selecting an appropriate k can also be an important issue because it can determine the tail latency and IOPS of i-NVMe. Figure 3 shows the performance variation as k increases. In these experiments, we create an environment where n is set to 2, and the interferer container runs 4 KB random read FIO workload without any rate limit. When k increases, the tail latency of the read request decreases to 20,000 μs when k is 32, as shown in Figure 3a, because requests can be handled without waiting for the next sub-quota. On the contrary, the IOPS in Figure 3b becomes lower when we increase k more than 16 due to frequent throttling overhead. This effect occurs when requests that could have been processed in a single sub-quota are processed in the next round by an excessively large k. In Figure 3b, when $k = 16$ or higher, the sub-quota was smaller than 1.56 ms, which is the I/O latency, so this trend was shown. It is noted that the degrading factor of the victim in Figure 3c is not changed much since k does not impact on the CPU usage of the victim. Therefore, the administrator must select k in consideration of the performance variation mentioned above according to the system's I/O requirements.

IV. Performance Evaluation

TABLE I: Evaluation Setup used in our evaluation

Hardware Configuration	
CPU	Intel Xeon Sliver 4210 CPU @ 2.20GHz
Memory	32GB
NIC	Mellanox ConnectX-5 EX (40G)
NVMe SSD	Samsung 970 Pro 512GB

Software Configuration	
OS	Linux 5.4.72
FIO	Block size=4KB, Direct I/O=on I/O engine=libaio, norandommap=on I/O depth=128

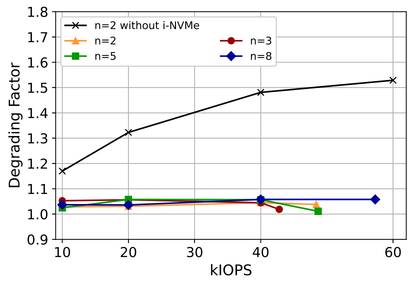


Fig. 4: Degrading Factor with multiple interferers.

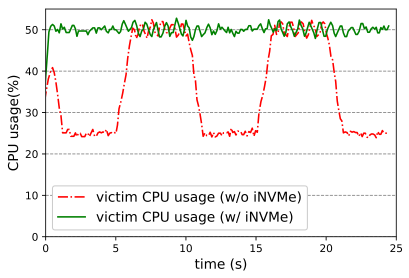
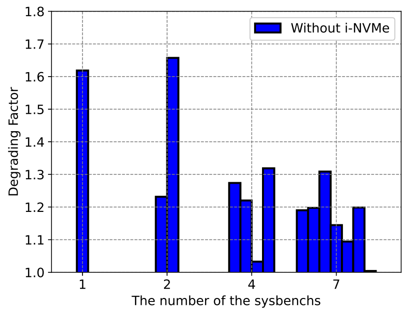
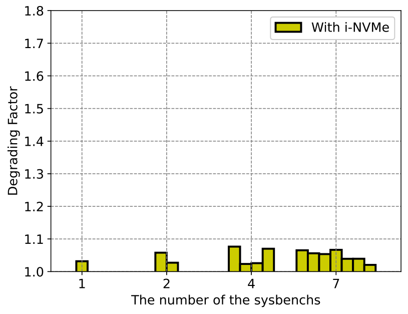


Fig. 5: victim's CPU usage with varying workload



(a) w/o i-NVMe



(b) w/ i-NVMe

Fig. 7: Degrading Factor with multiple victims and single interferer.

We implement i-NVMe on a real testbed and evaluate it with various remote I/O workloads. Note that all results are measured on the host side where containers are created.

A. Evaluation Setup

We set up NVMe over TCP testbed with two servers connected via a 40 Gbps link. Both servers have identical hardware and software setup as described in TABLE I except that the target node is equipped with an NVMe SSD. We use the FIO application with a default I/O depth (128) to generate NVMe over TCP workloads. Additionally, we enable direct I/O so that all requests bypass the operating system read and write caches and directly go through the network and to the remote SSD. All I/O requests are issued by the asynchronous I/O library in Linux (***libaio***).

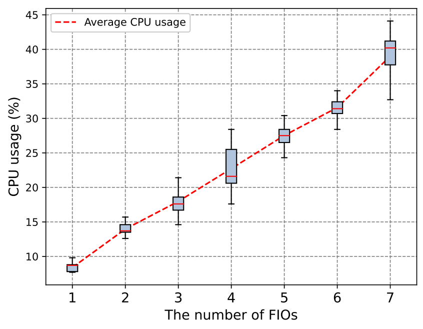
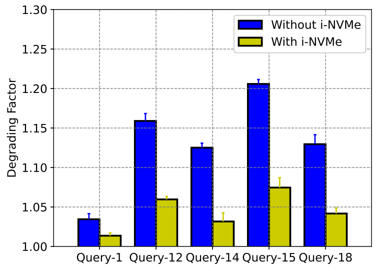
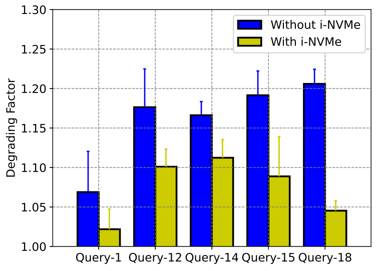


Fig. 6: CPU usage of *nvme_tcp_wq* when n is 8.

B. Benchmark Tests

We first describe how much the CPU isolation issue is improved by our scheme using benchmark workloads. Figure 4 shows the degrading factor of sysbench when we perform the same experiments as in Section II-C with our proposed scheme. Note that all experimental setups are the same as in Figure 1. It is shown that the degrading factor can be significantly improved with i-NVMe with various IOPS. While i-NVMe can effectively provide CPU isolation for victims, the performance of I/O intensive applications can be degraded. There are two causes of the performance degradation. One is that the overall CPU usage consumed by *nvme_tcp_wq* and the I/O application is reduced for providing CPU isolation, and this is inevitable for realizing isolation. The other one is that the CPU usage of *nvme_tcp_wq* is forcibly limited by i-NVMe. Without i-NVMe, *nvme_tcp_wq* can utilize the CPU with the highest priority, and it does not become the bottleneck of the I/O application. In i-NVMe, however, we have to decide how *nvme_tcp_wq* and the application share the given CPU time. If we allocate too much to *nvme_tcp_wq*, the application cannot issue enough I/O requests to consume the CPU usage of *nvme_tcp_wq*. If we allocate too much to the application, then *nvme_tcp_wq* becomes the performance bottleneck of the application. To observe how *nvme_tcp_wq* and the application share the CPU time in i-NVMe, we perform experiments with the following scenario. First, we run a FIO application with 40,000 IOPS workload without the CPU limit, and measure the CPU usage with `mpstat`. It is observed that the total CPU usage of *nvme_tcp_wq* and the application is 40%. Then, we run the next experiment with i-NVMe where the CPU time is limited by 40%, and measure the achieved IOPS. In


(a) victim is sysbench


(b) victim is wordcount

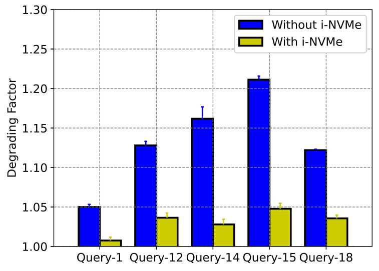

(c) victim is pi estimation

Fig. 8: Degrading Factor when a victim shares resources with MySQL.

the result, it is shown that the IOPS is in range between 38,600 and 39,500. This implies that i-NVMe can provide CPU isolation without serious performance degradation of I/O intensive applications by properly balancing CPU usages of *nvme_tcp_wq* and the application when the total CPU usage is identical.

Figure 5 shows in detail how our scheme preserves the victim's CPU quota. We measure the CPU usage changes of the victim's sysbench when FIO issues remote I/Os with maximum rates for five seconds and is paused for another five seconds, repeatedly. In the case of a normal Linux kernel without i-NVMe, the victim cannot utilize its quota when remote I/Os are issued as shown with the dotted line in Figure 5. It implies that the victim's CPU quota is interfered by FIO workload. With i-NVMe, the victim utilizes its CPU quota stably.

Figure 6 shows the variance of *nvme_tcp_wq*'s CPU usage when the number of interferers varies from one to seven when n is eight. The x-axis denotes the number of FIOs among eight containers, and the rest of containers run sysbench as victims. More FIO containers issue more remote I/Os, and io_used_quota increases with it. It makes a higher T value with our proposed adaptive threshold method resulting in an increase in CPU usage of *nvme_tcp_wq*.

Even if the number of victims increase, i-NVMe can preserve CPU isolation. In Figure 7, we run a single interferer with the maximum IOPS while varying the number of victim containers from one to seven, and measure the degrading factor of each victim container. Without i-NVMe, the degrading factors increase up to 65%, whereas i-NVMe preserves them lower than 7%. Note that the degrading factors of victims in Figure 7a tend to decrease as n increases. It is because that a less CPU quota is allocated to the single interferer with larger n, and the maximum IO usage and the CPU usage of *nvme_tcp_wq* also decrease.

C. Real Application Tests

To evaluate whether i-NVMe performs well with real applications, we run MySQL as an interferer. In Figure 8, the MySQL server is configured to store tables in the remote SSD. When queries are requested, I/O requests are generated, and *nvme_tcp_wq* is scheduled to handle them. Note that we use five TPC-H [16] sample queries, query types 1, 12, 14, 15, and 18 to generate I/Os. These query types consume sufficient CPU to fully utilize the MySQL container quota. Detailed explanations of sample queries are presented in [17]. For the victim, in addition to sysbench, we use two popular applications in distributed computing: `wordcount`, which counts the word occurrence frequency in a big file, and `pi estimation`, which computes the value of π.

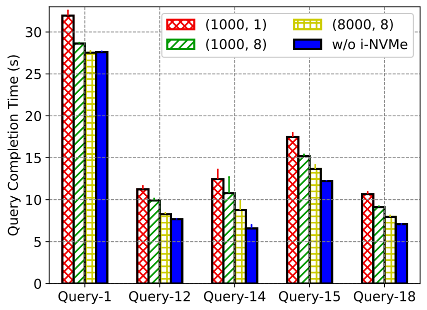


Fig. 9: TPC-H Query Completion time with various α and k (n is 2).

With sysbench in Figure 8a, our scheme increases the victim's completion time only by less than 7% for all query types, whereas it increases up to 21% without i-NVMe. In the case of query type 1, MySQL generates fewer I/O requests than the others, so that the degrading factor is lower than others. In Figures 8b and 8c, it is shown that the degrading factor can be efficiently contained with i-NVMe even when the interferer and the victim are real applications. In the case of `wordcount`, the average degrading factor of i-NVMe is slightly higher than other results since `wordcount` reads a file from the local file system, and unexpected contentions occur in the block layer.

D. I/O Performance Trade-off

We measure the completion time of each TPC-H sample query with various k and α pairs to understand the impact of i-NVMe on I/O performance with real applications. Other experimental setups are the same as those in Section IV-C. Figure 9 shows the average completion time of each TPC-H query type. The completion times of all query types decrease as both α and k increase owing to the shorter tail latency

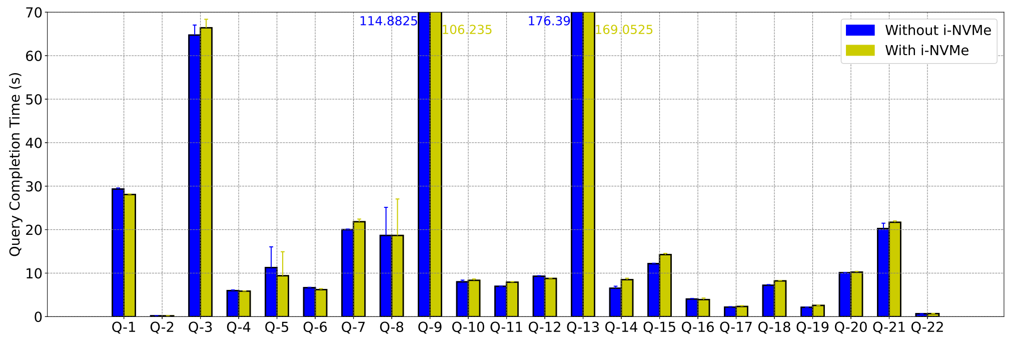


Fig. 10: TPC-H Query Completion Time for all sample query types (k, α is 8 and 8000 and n is 2).

and better I/O performance. It implies that a larger split parameter (k) and an increasing factor (α) can have positive effects on latency-sensitive applications. All parameter pairs in Figure 9 maintain the degrading factor within 7%. However, a large k ($k > 16$) decreases NVMe over TCP's IOPS as shown in Figure 3b. Therefore, the cloud administrator who applies i-NVMe should carefully adjust parameters according to the pattern required by the container's workloads, so that they can balance I/O performance-CPU isolation trade-off.

Lastly, Figure 10 shows the average completion time of all TPC-H query types with the parameter pair, $(k, \alpha) = (8, 8000)$, which is used for all other tests. Considering the result of the figure and other results together, we can conclude that i-NVMe provides a high level of CPU isolation without significant performance degradation for real-world applications that do not use heavy I/O.

V. Related Work

In recent years, there have been several studies to optimize or re-design the NVMe over TCP stack in performance perspective such as Reflex [18] and i10 [19]. ReFlex introduced the user-level remote flash access stack, and it was shown that ReFlex can generate a high I/O throughput with low latency; however, modification of existing applications is inevitable. In addition, the use of such user space stacks is not suitable for containerized environments in which various users and workloads exist. It prevents other applications from using the normal kernel stack.

i10 provided high-performance TCP-based remote storage stacks without user-level protocols or additional expensive hardware. Through optimizations within the Linux kernel such as resource dedication in a proper manner and batching, i10 shows a similar performance to NVMe over RDMA. Despite the i10's performance improvement, batching without considering the containerized environments can cause additional interference issues. As the CPU usage consumed by i10 itself is not properly charged, the problem pointed out in our study is likely to appear. The performance of the TCP based remote storage stack is continuously increasing, but it is hard to exploit them in the containerized environments. It is because they are not compatible with different types of containers, and the performance interference is not considered.

Many studies pointed out the interference problems when processes share resources and suggested resource isolation schemes to resolve them. Especially, some of them have investigated ways to isolate each process or group's CPU usage [9], [20], [21], network bandwidth [22]–[24], and storage resource [25]–[27]. These studies are effective to allocate and measure each container's resource usage and isolate them, but they cannot handle indirect CPU usages caused by kernel workers as pointed out in this study.

The indirect resource usages caused by different types of kernel stack processing were also discussed in several previous studies. IRON [13] handled the preempted CPU usage caused by network interrupts. mClock [28], SplitIO [26] measured block I/O processing overhead and designed tagging schemes for accounting. They effectively measured these hidden costs in a per-container manner. But, they only considered the resource usages consumed by specific containers and the stack processing costs of directly generated traffic, so they are not applicable to NVMe over TCP.

VI. Conclusion

This study proposed i-NVMe to resolve the CPU isolation violation of NVMe over TCP. We measured, charged the CPU usages consumed by NVMe over TCP proportional to each container's I/O usage, and throttled its traffic if necessary. With carefully selected parameters, we maintained the balance between i-NVMe's performance and CPU interference. Our extensive experiments on the real testbed showed that i-NVMe significantly lowered CPU interference caused by indirect CPU usages for handling NVMe over TCP I/O traffic, and CPU isolation was successfully enforced among CPU-intensive and data-intensive containers.

Acknowledgment

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT). (NRF-2020R1A2C2009809, NRF-2021R1C1C1009778.) We would like to thank the anonymous reviewers for their insightful comments and suggestions.

References

[1] J. Taylor, "Facebook's data center infrastructure: Open compute, disaggregated rack, and beyond," in *Optical Fiber Communication Conference*. Optical Society of America, 2015, pp. W1D–5.

[2] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes, "Building an elastic query engine on disaggregated storage," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 449–462.

[3] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia, "A remote direct memory access protocol specification," RFC 5040, October, Tech. Rep., 2007.

[4] I. T. Association *et al.*, "Infinibandtm architecture specification volume 1 release 1.3 (general specifications)," 2015.

[5] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 202–215.

[6] 100g kernel and user space nvme/tcp using chelsio toe. [Online]. Available: https://www.chelsio.com/wp-content/uploads/resources/t6-100g-spdk-nvmetoe.pdf

[7] Nvme over tcp test report with the mellanox toe. [Online]. Available: https://community.mellanox.com/s/article/NVMe-over-TCP-Test-Report

[8] L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three cpu schedulers in xen," *Performance Evaluation Review*, vol. 35, no. 2, p. 42, 2007.

[9] P. Turner, B. B. Rao, and N. Rao, "Cpu bandwidth control for cfs," 2010.

[10] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim, "Eyeq: Practical network performance isolation at the edge," in *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, 2013, pp. 297–311.

[11] L. Cheng and C.-L. Wang, "Network performance isolation for latency-sensitive cloud applications," *Future Generation Computer Systems*, vol. 29, no. 4, pp. 1073–1084, 2013.

[12] Linux control group block io controller. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt

[13] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella, "Iron: Isolating network-based {CPU} in container environments," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 313–328.

[14] Linux workqueue documentations. [Online]. Available: https://www.kernel.org/doc/html/latest/core-api/workqueue.html

[15] J. Axboe. (2019) Flexible io tester (fio) ver 3.13. [Online]. Available: https: //github.com/axboe/fio

[16] Tpc-h benchmarks. [Online]. Available: http://www.tpc.org/tpch/

[17] Tpc-h sample query documentations. [Online]. Available: https://docs.deistercloud.com/content/Databases.30/TPCH Benchmark.90/Sample querys.20.xml

[18] A. Klimovic, H. Litz, and C. Kozyrakis, "Reflex: Remote flash=local flash," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 345–359, 2017.

[19] J. Hwang, Q. Cai, A. Tang, and R. Agarwal, "{TCP}≈{RDMA}: Cpu-efficient remote storage access with i10," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 127–140.

[20] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi2: Cpu performance isolation for shared compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 379–391.

[21] D. B. Bartolini, F. Sironi, D. Sciuto, and M. D. Santambrogio, "Automated fine-grained cpu provisioning for virtual machines," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 3, pp. 1–25, 2014.

[22] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, and C. Kim, "Eyeq: Practical network performance isolation for the multi-tenant cloud," in *4th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 12)*, 2012.

[23] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "Faircloud: Sharing the network in cloud computing," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, 2012, pp. 187–198.

[24] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos, "Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, 2013, pp. 351–362.

[25] L. Lu, Y. Zhang, T. Do, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Physical disentanglement in a container-based file system," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 81–96.

[26] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. Al-Kiswany, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Split-level i/o scheduling," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 474–489.

[27] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "Ioflow: A software-defined storage architecture," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 182–196.

[28] A. Gulati, A. Merchant, and P. J. Varman, "mclock: Handling throughput variability for hypervisor io scheduling." in *OSDI*, vol. 10, 2010, pp. 437–450.