Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Performance Characterization Study of NVMe Storage Over TCP

Author: Sudarsan Sivakumar (2722524)

1st supervisor:	Tiziano De Matteis
daily supervisor:	Krijn Doekemeijer, Animesh Trivedi
2nd reader:	Balakrishnan Chandrasekaran

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

October 22, 2024

“Computer science is operating system for all innovation”

, by Steve Ballmer

Abstract

Storage is a critical component of Internet services, and as the reliance on cloud-based services continues to grow, these services demand high performance from cloud providers. In cloud environments, storage is often disaggregated, where storage and compute resources are separated. While this provides greater flexibility and scalability, it can also introduce performance overheads due to increased network transmission and protocol processing.

Additionally, cloud vendors share resources among multiple users to improve utilization, which leads to performance interference. This interference can significantly impact the performance of cloud storage, especially in disaggregated storage systems where various resources, such as compute, network, and storage, are involved.

This study investigates and quantifies the performance overheads and interference in NVMe over Fabrics using TCP (NVMeoF-TCP), a protocol designed for block-level storage disaggregation. Through experimentation, we analyze how different NVMeoF-TCP configurations affect system performance. Our findings indicate that optimizing default settings can achieve latency reductions of up to 19% and throughput improvements of up to 14.4%. However, compared to local storage, NVMeoF-TCP introduces a 27.3% increase in latency and reduces single-core throughput by 2.58 times. Additionally, our examination of varying background loads shows that different components of the NVMeoF-TCP stack respond uniquely to these conditions, with latency interference ranging from 2 to 38 times, depending on the load experienced by the initiator and target cores. These insights are crucial for optimizing NVMeoF-TCP deployments and enhancing performance in disaggregated storage environments, paving the way for more efficient and scalable solutions.

The artifacts of this thesis are available online on GitHub at https://github.com/bergstartup/NVMeoFTCP-Characterization

Contents

List of Figures

List of Tables

LIST OF TABLES

1

Introduction

In the Netherlands, data centers and related ICT infrastructure play a critical role in the economy, supporting over 3.3 million jobs and contributing to more than 60% of the GDP(1). With such a significant economic impact, ensuring the cloud infrastructure operates smoothly and efficiently is vital. Cloud infrastructure powers most internet services, offering the scalability and flexibility needed for many applications (2). Companies like Netflix and Amazon rely on delivering smooth, efficient user experiences, as their revenue is closely tied to performance. Faster page loads, seamless video streaming, and quick transaction processing are vital to keeping users satisfied. A Deloitte study reveals that improving website load times by just one millisecond can increase conversion rates by 8-10% for retail and travel sites (3). Cloud users enter into Service Level Agreements (SLAs) that specify expected service performance to ensure consistent performance.

Cloud providers maximize resource usage by sharing infrastructure among multiple users, a practice known as multitenancy(4). However, multitenancy can result in performance variability, as different users compete for shared resources(5). This variability poses a challenge to meeting the performance standards of the SLAs. Therefore, cloud providers are responsible for ensuring their services remain performant and minimise interference to meet SLA requirements.

Storage is a critical part of cloud infrastructure. Cloud providers use disaggregation, where storage is separated from compute resources. Storage requests from compute nodes are sent over the network to storage servers, making it easier to scale both storage and compute independently. This also helps reduce the Total Cost of Ownership (TCO) for providers (6, 7, 8). However, disaggregation can introduce performance overhead due to transport protocol processing and network transmission. Additionally, performance interference can arise from the multiple resources involved in disaggregated storage systems.

This thesis aims to quantify the performance overhead and interference associated with storage disaggregation in cloud environments. We focus on NVMe over Fabrics using TCP (NVMeoF-TCP), a protocol designed for block-level disaggregation. Our results show that, compared to local storage, NVMeoF-TCP introduces a 27.3% increase in latency and reduces single-core throughput by 2.58 times. The interference analysis reveals that different components of the NVMeoF-TCP stack respond differently to varying loads, with performance interference ranging from 2x to 38x depending on the component and load conditions.

1.1 Context

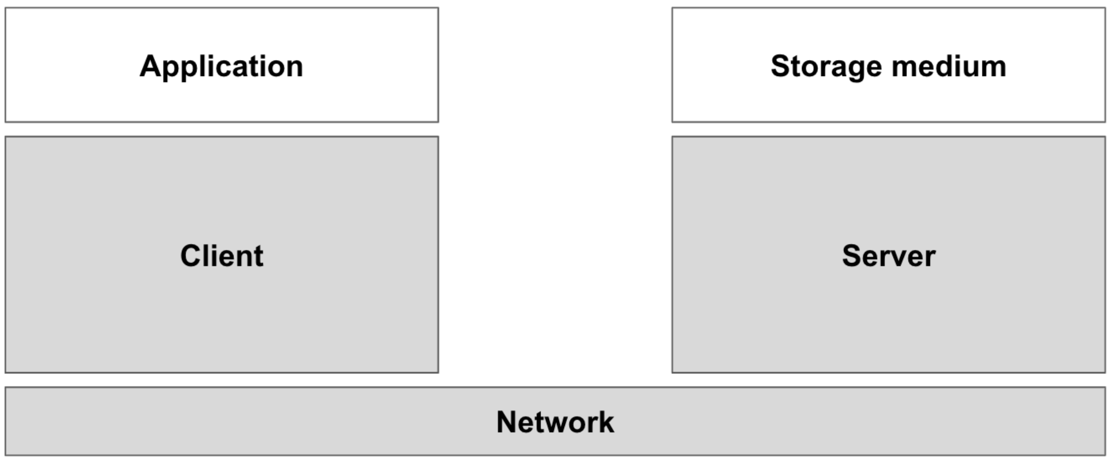

Figure 1.1: General architecture of disaggregated storage

Typically, remote storage has four hardware components, as shown in Fig1.1: the client where the applications that issue I/O runs, the network through which the I/O requests are transferred, the server where the I/O requests from the client are processed, and the storage medium. A remote storage protocol binds all of them together. The remote storage protocols determine the transport protocol used and the layer of disaggregation. Usually, the disaggregation happens either at the file layer or the block layer.

In this thesis, we investigate NVMeoF-TCP, a block layer disaggregation protocol that uses the TCP transport protocol to access NVMe storage devices remotely. Though there are several protocols for storage disaggregation, we choose NVMeoF-TCP specifically for the following reasons:

- Wide Compatibility: NVMeoF-TCP operates over standard TCP/IP networks, making it compatible with the existing network infrastructure of cloud providers.
- Performance: NVMeoF-TCP is an extension of NVMe, a high-performance protocol for accessing local flash storage devices over PCIe. NVMe has become a standard for fast, low-latency access to flash storage.

1.2 Problem Statement

Storage disaggregation in NVMeoF-TCP can lead to performance overheads caused by additional network transmission and extra processing within the software stack. Furthermore, performance interference may occur due to resource contention within the NVMeoF-TCP stack. Therefore, it is essential to quantify the performance overheads and the interference effects from various resources within the NVMeoF-TCP stack to optimize overall system performance.

1.3 Research Questions and Methodology

To solve the problems mentioned previously, we present the following research questions

- **(RQ1) What are the performance implications of NVMeoF-TCP polling configurations?**
  Misconfiguration is a common system issue and can lead to suboptimal performance (9). Proper configuration of NVMeoF-TCP is essential for achieving optimal performance. We specifically focus on polling configurations because they directly impact how I/O requests are processed, affecting latency and throughput. We aim to identify configurations that can improve performance and reduce processing delays by analysing different polling settings. We make use of the following research methodologies to answer this research question:
  - (M1) Experimental research: designing appropriate micro-benchmarks to quantify a system (10).
  - (M2) Open science, open source software, community building, reproducible experiments(11, 12).
- **(RQ2) What is the performance overhead of NVMeoF-TCP compared to local NVMe?**

Building on the findings from RQ1, we compare the performance overhead of NVMeoF-TCP to local NVMe, focusing on key metrics such as latency, throughput, and bandwidth. To address RQ2, we employ methodologies M1 (experimental research) and M2 (open science practices) for a thorough and reproducible evaluation.

- **(RQ3) What is the performance interference with various loads on compute resources in the NVMeoF-TCP stack?**
  We study how varying levels of compute load affect performance interference within the NVMeoF-TCP stack. We do not include network resources, as our setup cannot generate enough load for useful interference tests, and we exclude SSDs because their interference behaviour varies too much between SSDs. To answer RQ3, we use methodologies M1 (experimental research) and M2 (open science practices).

1.4 Thesis Contributions

This thesis makes three contributions. First, we provide different configuration parameters that can positively affect the NVMeoF-TCP performance. Second, we quantify the performance overhead of NVMeoF-TCP over local NVMe. Third, we also quantify the performance interference in the compute resources of the NVMeoF-TCP stack.

1.5 Societal Relevance

By 2025, it is estimated that the cloud industry will consume 20% of global electricity and contribute 5.5% of global carbon emissions(13). In this thesis, we aim to improve the performance of disaggregated storage and analyze resource interference, helping cloud providers make better resource-sharing decisions. This can lead to more efficient resource utilization and ultimately reduce carbon emissions.

1.6 Thesis Structure

In Chapter 2, We provide the necessary background on Disaggregated storage, NVMeoF, and NVMeoF-TCP internals. In Chapter 3, We present the experiment setup and configure the parameters of NVMeoF-TCP for better performance. In Chapter 4, We compare the performance of NVMe and NVMeoF-TCP to answer the RQ1. In Chapter 5, We quantify the interference caused under loads for different resources in NVMeoF-TCP. In Chapter 6, we discuss the limitations of the current work and the related works. Finally, in Chapter 7, We conclude by summarizing the findings of this thesis, answering the research questions, and providing possible future works that could be developed from the findings of this thesis.

Plagiarism Declaration

I confirm that this thesis work is my work, is not copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment. I understand that plagiarism is a serious issue and should be dealt with if found.

2

Background

2.1 Disaggregated Storage

Disaggregated storage refers to separating the storage of data from the computing. This means that instead of having storage built directly into each server, the storage is placed separately and connected to the servers through a network. This approach allows easier scaling, management, and flexibility for the allocation of resources. Disaggregated storage can be implemented at different layers, most commonly at the file and block levels.

2.1.1 File-based Disaggregation

File-based disaggregation occurs at the file system level. In this model, storage is accessed remotely via a shared file system protocol such as NFS (Network File System) or distributed systems like HDFS (Hadoop Distributed File System). Clients request access to entire files, which are managed and served by a central server or a cluster of storage nodes.

2.1.2 Block-based Disaggregation

Block-based disaggregation operates at a lower layer by providing remote access to storage at the block level. The common protocols in block disaggregation are iSCSI (Internet Small Computer System Interface), FCoE (Fibre Channel over Ethernet), and NVMeoF (Non-Volatile Memory Express over Fabrics).

2.2 Non-Volatile Memory Express (NVMe)

NVMe (Non-Volatile Memory Express) is a high-performance storage protocol over PCIe designed to maximize the potential of modern non-volatile memory technologies, such as

SSDs. NVMe achieves better performance compared to protocols like SATA and SAS by exploiting the parallelization of SSD-based storage devices.

NVMe over Fabrics

NVMe over Fabrics (NVMeoF) is an extension of NVMe for disaggregated block storage. NVMeoF protocol allows NVMe commands to be transmitted over a network fabric, making remote NVMe devices accessible with similar performance characteristics to local NVMe storage. The architecture of NVMeoF consists of three components: the initiator, the target, and the transport protocol. The initiator hosts the application and sends I/O requests over the network using the transport protocol to the target, while the target hosts the storage device(s) and processes these I/O requests. The NVMeoF supports the following transport protocols:

- Fibre Channel
- RDMA
- **TCP** (focus of this thesis)

2.3 NVMeoF-TCP

We examine the functioning of NVMeoF-TCP by dividing it into two main components: the control plane and the data plane. In the control plane, we discuss the steps involved in the initialization of an NVMeoF-TCP remote device, including setting up the target, configuring the transport layer, and establishing the connection between the initiator and target, along with mapping the namespaces. On the other hand, in the data plane, we explore queue management and the process of I/O transmission. This includes how NVMeoF-TCP manages queue pairs for each core and processes I/O requests and responses between the initiator and target. Additionally, we detail how worker functions handle I/O processing for both the initiator and the target.

NVMeoF-TCP controlplane

On the target side, any NVMe-based storage devices (marked as ❶ in Fig. 2.1) that need to be accessed by the initiator must be registered with the NVMeoF target module. To initialize the NVMeoF module, a port must first be registered. This port contains configuration details such as the transport type (which is TCP in our case), transport

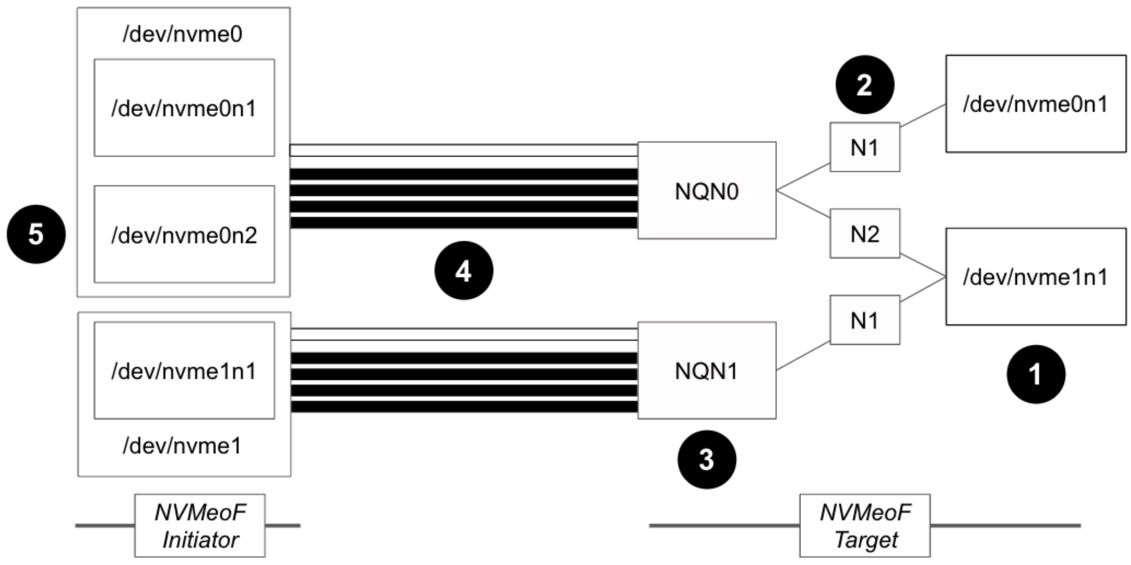

Figure 2.1: NVMeoF-TCP control plane

endpoint details (IP address and port), and access specifications (e.g., "Allow all"). Once the port is registered, the next step is to create a NQN (NVMe Qualified Name) 3 and map the SSD devices to namespaces 2.

Each NQN acts as a separate NVMeoF controller, anNVMeoF-TCP-CPd while multiple SSD devices can be mapped under the same NQN, they must be assigned to different namespaces to ensure proper functionality. On the initiator side, to access these remote block devices, the target address (IP and port) and the corresponding NQN must be specified using the nvme connect command, which is part of the NVMeCLI tool(14). Once the connection to an NQN is successfully established, all SSD devices associated with that NQN will appear on the initiator as remote block devices (e.g., /dev/nvmeXn1, /dev/nvmeXn2...) 5.

By default, each NQN connected from the initiator creates N+1 TCP connections to the target, where N is the number of cores in the initiator 4. The number of TCP connections is controlled by the number of queue pairs, which will be discussed in the upcoming section.

NVMeoF-TCP dataplane

Queue management

When a remote NVMeoF-TCP device is initialized, a queue pair is created for each core in the initiator. For every queue pair, a corresponding TCP connection is established between

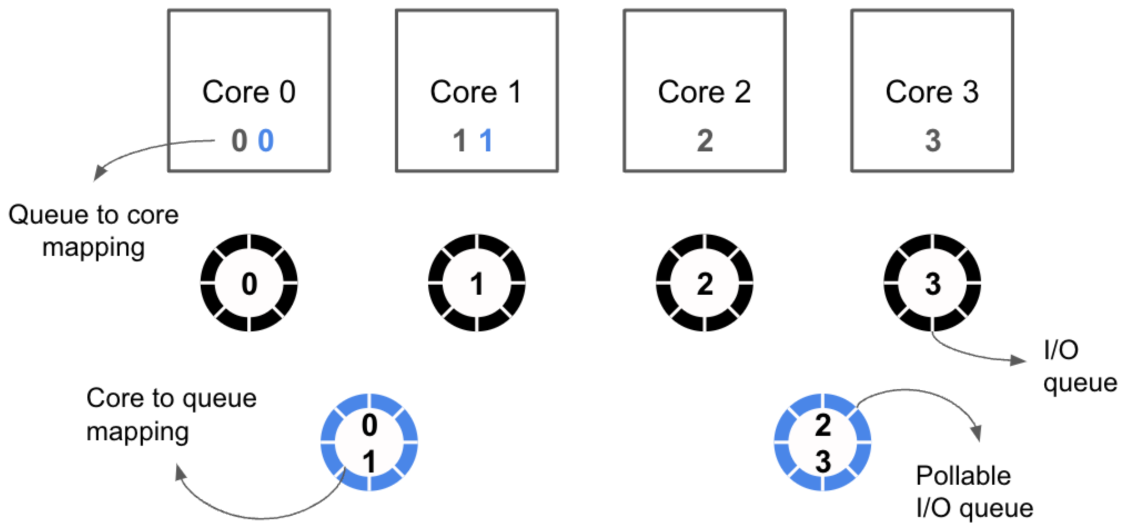

Figure 2.2: I/O queue pair and core mappings in the initiator of the NVMeoF-TCP

the initiator and the target. On the target side, each of these TCP connections generates an NVMeoF-TCP target queue pair, which acts as a buffer. This buffer is used to receive requests from the initiator and to handle responses from the local NVMe storage devices.

Worker function

A worker function in the NVMeoF-TCP stack is a callback function responsible for processing a command (CMD) or response that has been added to the corresponding NVMeoF-TCP queue pair. Each queue pair has its separate worker function. The worker function can be invoked either synchronously or asynchronously:

- Synchronous call: If invoked synchronously, the worker function is executed in the same process context that inserted the element into the queue. This means the process directly calls the function, handling the command/response without deferring execution.

- Asynchronous call: If invoked asynchronously, the execution of the worker function is deferred through the work queue interface. This allows the system to schedule the execution of the function at a later time, reducing the immediate load on the initiating process. However, the work queue has a constraint: it allows only one entry per worker function, even if multiple asynchronous invocations occur. This ensures that only one instance of the worker function is processed at any given time.

There are separate worker functions for the initiator and target sides of the NVMeoF-TCP system:

- Initiator worker function: This function alternates between processing one element from the send queue (handling an NVMeoF-TCP request) and one from the receive queue (handling an NVMeoF-TCP response). The function runs for a maximum of 1 millisecond in each call (a hardcoded limit). If there are still elements left in the queues when the worker finishes, it requeues itself in the work queue for later execution.

- Target worker function: The target worker function processes 8 responses and 8 requests in an alternating fashion. During a single execution context, it processes up to 64 elements from both the send and receive queues. These limits are also hardcoded. If there are still pending requests or responses in the queues when the function completes, it requeues itself in the work queue to continue processing.

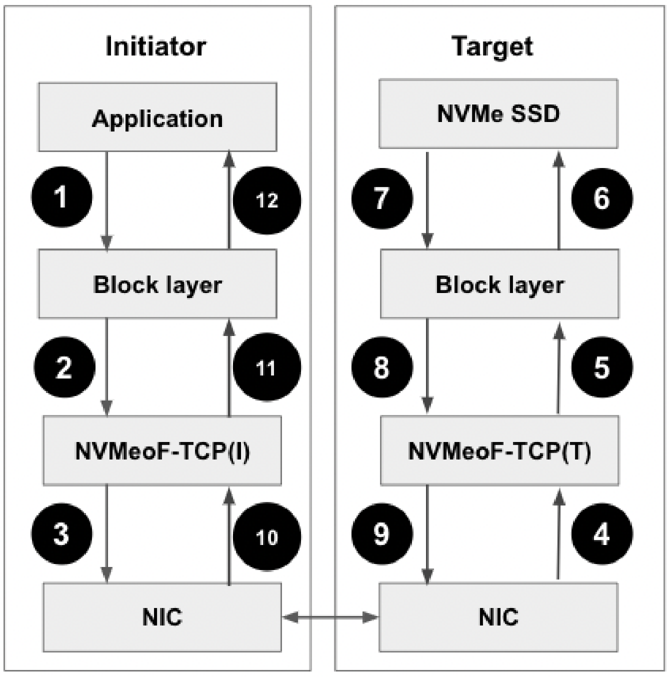

Figure 2.3: Request traversal in NVMeoF-TCP stack

I/O transmission tracing in NVMeoF-TCP stack

We will be tracing the I/O request submission and response in NVMeoF-TCP (Visually represented in Fig2.3):

1. (Application to Block layer) The application submits a request to the block layer queue via a system call.

2. (Block layer to NVMeoF-TCP initiator) In the same process context as the system call, the block layer request is converted into an NVMeoF-TCP CMD PDU and added to the NVMeoF-TCP send queue.

3. (NVMeoF-TCP initiator to NIC) In the same context, the NVMeoF-TCP initiator worker function processes the CMD PDU and performs TCP/IP processing. If there is no packet backlog, it transmits directly to the NIC, which then forwards the packet to the target.

4. (NIC to NVMeoF-TCP target) Upon receiving the packet, the target NIC places it in the RX queue and triggers an interrupt, which schedules the NET_RX_SOFTIRQ. This performs TCP/IP processing and places the NVMeoF-TCP CMD PDU into the NVMeoF-TCP target queue, enqueuing the target worker function into the work queue.

5. (NVMeoF-TCP target to Block layer) The target worker function reads the CMD PDU from the socket buffer, creates a block request and submits it to the block plug.

6. (Block layer to NVMe SSD) The block plug is flushed, sending the request to the NVMe SSD.

7. (NVMe SSD to Block) When the NVMe SSD completes the request, it triggers an interrupt that schedules a BLOCK SOFTIRQ.

8. (Block layer to NVMeoF-TCP target) The BLOCK SOFTIRQ queues the response in the NVMeoF-TCP target queue and enqueues the target worker function into the work queue.

9. (NVMeoF-TCP target to NIC) The target worker function dequeues the response, processes it with TCP/IP, and transmits it directly to the NIC if there is no backlog. The packet is then sent from the target to the initiator.

10. (NIC to NVMeoF-TCP initiator) Interrupts scheduler NET_RX_SOFTIRQ. NET_RX_SOFTIRQ processes TCP/IP and schedules the initiator worker function.

11. (NVMeoF-TCP initiator to block) The initiator worker function schedules the BLOCK SOFTIRQ.

12. (Block to application) The BLOCK SOFTIRQ sends the response back to the application.

3

Performance Characterization Setup

In this chapter, we specify the setup and configurations for the experiments and provide the evaluation plan for the upcoming chapters.

3.1 Experiment Setup

We set up QEMU virtual machines (VMs) for the initiator and target on host machines configured as specified in Table 3.1. The host machines for the initiator and target are connected via a 100 Gbps Ethernet link. All the peripherals listed in Table 3.2 are passed through to the corresponding QEMU VMs using VFIO. All peripherals, VM processes, and memory are pinned to the same NUMA domain on the host machines to ensure optimal performance. The network interface card (NIC) is connected to NUMA domain 0 on the initiator's host machine. Therefore, we use the numactl command to ensure that the memory and processes of the initiator VM are also linked to NUMA domain 0. Similarly, the NIC and SSDs are located in NUMA domain 1 on the target host machine, so we use numactl to attach the target QEMU VM to that domain.

Component	**Initiator**	**Target**
Kernel	Linux 6.8	Linux 6.8
Operating System	Ubuntu 24.04 LTS	Ubuntu 24.04 LTS
CPU	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
Number of Core(s)	10	10
RAM	24 GB	24 GB
MTU	9000 bytes	9000 bytes

Table 3.1: System software configuration of initiator and target

Component	**Initiator**	**Target**
NIC	Mellanox ConnectX-5	Mellanox ConnectX-5
SSD	-	2 x Western Digital SN540 SSDs

Table 3.2: Peripherals configuration of initiator and target

Regarding the peripherals' performance, the local tail latency for random read operations on the SSD is 83.5 microseconds, supporting up to 459 KIOPS. For the 100 Gbps link between the initiator and target, we achieved a bandwidth of 97 Gbps using the TCP_BW test with iperf3.

We used Fio version 3.37 for all our storage benchmark experiments. Unless stated otherwise, the following were the common configurations across all experiments:

- Ramp-up time: 60 seconds
- Run time: 120 seconds
- Direct I/O: Enabled
- IO engine: io_uring (hipri = 1, registerfiles = 1)
- Workload: Random read (randread)

Each experiment in this chapter was run three times, and we report the mean values of the metrics across runs to ensure better confidence in our observations.

Hypothesis: Impact of NET RX Interrupt Affinity and Block Completion Polling on Initiator Performance

Interrupt affinity and completion polling play an important role in the performance of NVMeoF-TCP at the initiator (15). These parameters directly affect how response processing is managed, which occurs in two main steps:

1. TCP/IP processing happens in the NET_RX_SOFTIRQ part.
2. The worker function runs on the CPU core until the application(fio) gets the response.

Interrupt affinity refers to the CPU core that handles an interrupt from a device. Here, we're looking at the interrupt affinity for network receive (NET RX) interrupts from a network card, which further dictates the corresponding NET_RX_SOFTIRQ core placement (Part 1 in the response processing). Making both the NET_RX_SOFTIRQ processing and the worker function (Part 2) run on the same core helps cache locality.

The worker function's core placement depends on where the corresponding application thread is running, while the core for NET RX depends on which receive (RX) queue the network card places the TCP packets of the corresponding NVMeoF-TCP response. To align the cores of the two parts, we can use Accelerated Receive Flow Steering (aRFS). This feature allows the operating system to give the network card hints on which CPU core to use based on where the application thread runs.

The next system configuration we are concerned with is application polling for block completions. Polling alleviates the overhead associated with multiple context switches during interrupt handling. We can enable polling for our experiment by setting the hipri argument while the io_engine is set to io_uring in the fio job file.

Intuitively, These system configurations can interfere with each other. So, we experimented to identify the interaction of these two configurations.

p99 latency(μs)/KIOPS	**With polling**	**Without polling**
With interrupt affinity	106.66/122.76	121.34/118.70
Without interrupt affinity	102.22/124.38	124.41/112.79

Table 3.3: Performance matrix for polling(Block layer) and interrupt affinity(NET_RX_SOFTIRQ)

Based on the Table3.3, polling without interrupt affinity performs better than other configurations. So, we choose to disable aRFS(no interrupt affinity) and enabled hipri=1 for completion polling.

3.2 Application Modelling and Corresponding Performance Metrics

In this section, we will define different types of application models and the corresponding performance metrics that will be used in the upcoming chapters to evaluate performance.

- **L-app (Latency-sensitive applications):** This type of application is represented by the fio configuration: P1, QD1 (Queue Depth 1), and Block size 4KB. These applications submit a single outstanding request and expect quick completion. The performance metric used to evaluate L-app performance is p99 latency, which reflects the 99th percentile of request completion times. Examples of latency-sensitive applications include online gaming platforms and real-time financial trading systems, where low latency is crucial for user experience and decision-making.

- **T-app (Throughput-sensitive applications):** The corresponding fio configuration for this type is P1, Block size 4KB, with varying Queue Depths. These applications submit multiple concurrent requests and aim to complete as many as possible in a given time. The performance metric for T-app is throughput, measured in IOPS. Common examples of throughput-sensitive applications are large-scale database management systems and cloud storage services, which handle many requests simultaneously.

- **B-app (Bandwidth-sensitive applications):** This application is characterized by the fio configuration: P1, varying block sizes and queue depths. These applications require the reading and writing large volumes of data, and their performance is measured in terms of bandwidth, typically expressed in MB/s. Examples of bandwidth-sensitive applications include video streaming platforms and big data analytics, where transferring and processing large amounts of data is essential for performance.

4

Configuring NVMeoF-TCP for Performance

In this chapter, we answer **(RQ1) What are the performance implications of NVMeoF-TCP polling configurations?** We start by identifying all the configuration options available for NVMeoF-TCP and then focus on the ones that use polling. After selecting the polling configurations, we evaluate their performance to see how they affect the system. From this analysis, we choose the best-performing polling setup, which will be used for further experiments to answer other research questions.

4.1 Identifying NVMeoF-TCP Configuration Knobs

NVMeoF-TCP offers configuration options through module parameters and per NQN (Qualified Name) parameters, which the initiator specifies during connection initialization. The module parameters apply to the entire module and are enforced across all connections and remote devices, while the NQN parameters are specific to each remote device. The module and NQN parameters include the following:

- **Target idle polling (Target module parameter)**: Determines how long the NVMeoF target worker function is requeued in the work queue without a new request to process. This is referred to as idle polling time. The default is 0, meaning the target worker function is not rescheduled if no additional requests or responses are pending.

- **Number of NVMeoF I/O queues (NQN parameter)**: Defaults to the number of cores on the initiator.

- **Polling I/O queues (NQN parameter)**: This creates queue pairs with pollable completion queues in the initiator. The default number of pollable queues is 0.
- **Socket priority (Initiator and Target module parameter)**: Specifies the network packet priority for all NVMeoF-TCP flows from that node.
- **Type of service (NQN parameter)**: Similar to socket priority but is specified for each NQN, applying to both the initiator and target node for all connections related to that NQN.

4.2 Evaluating NVMeoF-TCP Polling Configuration Knobs for Performance

From the listed parameters, we select those that use a polling mechanism. The chosen parameters are:

1. Polling I/O queues
2. Target idle polling

These configurations are evaluated incrementally to observe their effects on performance.

Initiator completion polling using polling I/O queues

In the default configuration, the polling kthread calls the bio_poll function, which looks for block I/O (bio) completion events. If pollable I/O queue pairs are enabled for NVMeoF-TCP, the bio_poll function further invokes the nvme_tcp_poll function, which checks for packet reception on the socket associated with the queue pair as shown in Fig4.1. Polling the NVMeoF-TCP queues removes the need to execute the initiator worker function. Intuitively, this polling approach could enhance performance by eliminating context switches associated with the initiator worker execution.

Configuration	**Throughput (KIOPS)**	**p99 latency (μs)**
Default configuration	114.22	127.14
Pollable I/O queues	130.73	115.20

Table 4.1: Throughput and tail latency for default and pollable I/O queues configurations

To evaluate the improved performance claim, we run an experiment comparing the performance of completion polling with pollable I/O queues against the default configuration.

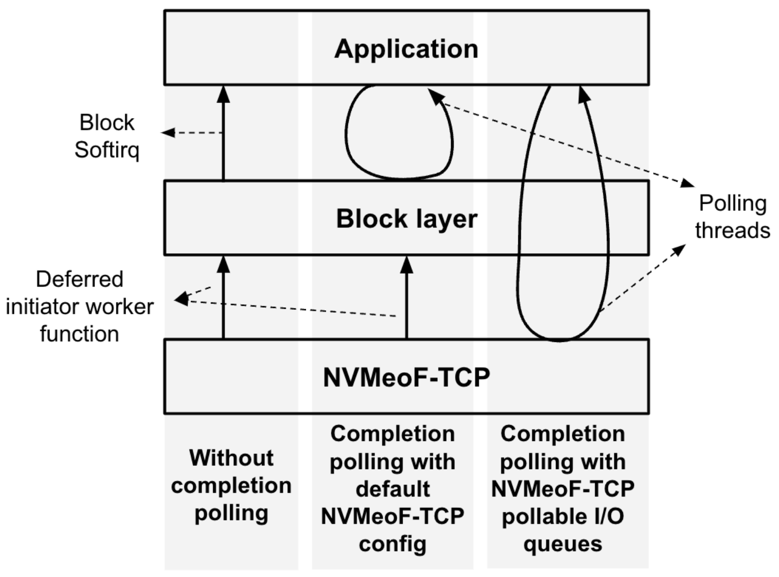
Application
Block Softirq
Block layer
Polling threads
Deferred initiator worker function
NVMeoF-TCP
Without completion polling
Completion polling with default NVMeoF-TCP config
Completion polling with NVMeoF-TCP pollable I/O queues

Figure 4.1: Different completion semantics for NVMeoF-TCP

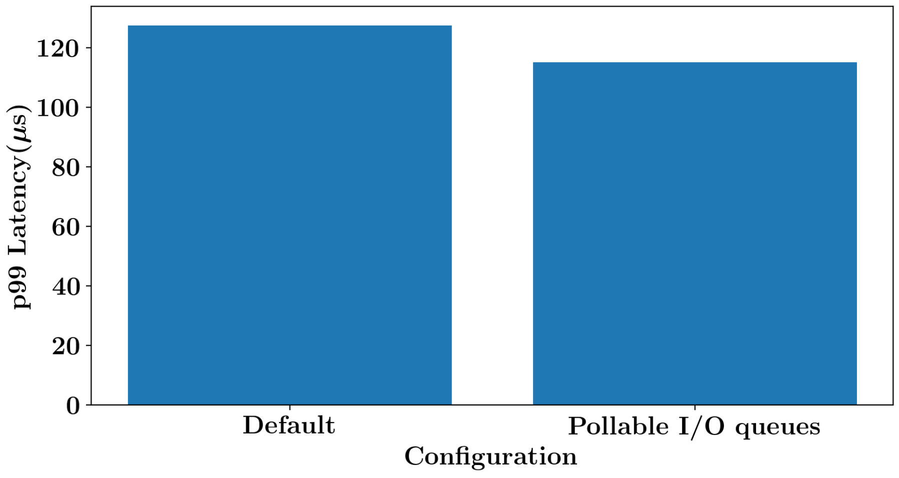

Figure 4.2: Plot for p99 latency for default and pollable I/O queues configurations

The results are presented in Fig. 4.2 for p99 latency and Fig. 4.3 for throughput. The following observations can be drawn from the results:

1. In terms of p99 latency, pollable I/O queues shows a 9.3% (11.94μs) improvement compared to default configuration.
2. For maximum achieved IOPS from a single core, pollable I/O queues deliver a 14.4% improvement over the default configuration.

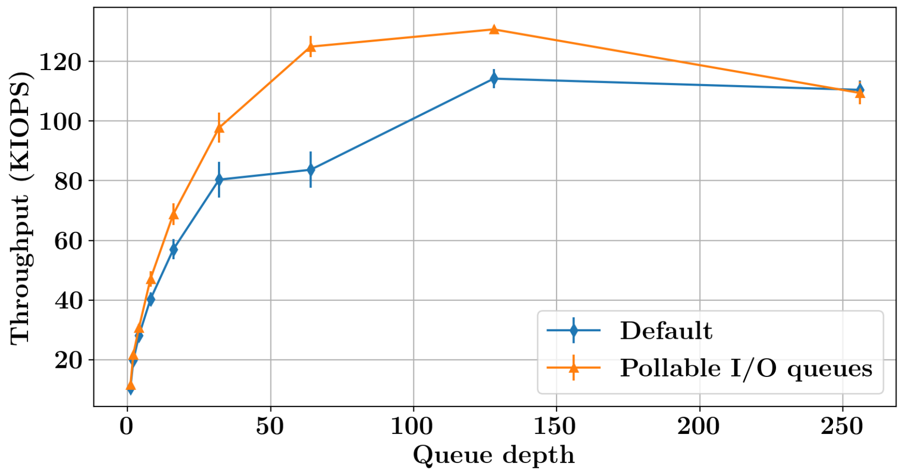

Figure 4.3: Plot for IOPS vs Queue depth for default and pollable I/O queues configurations

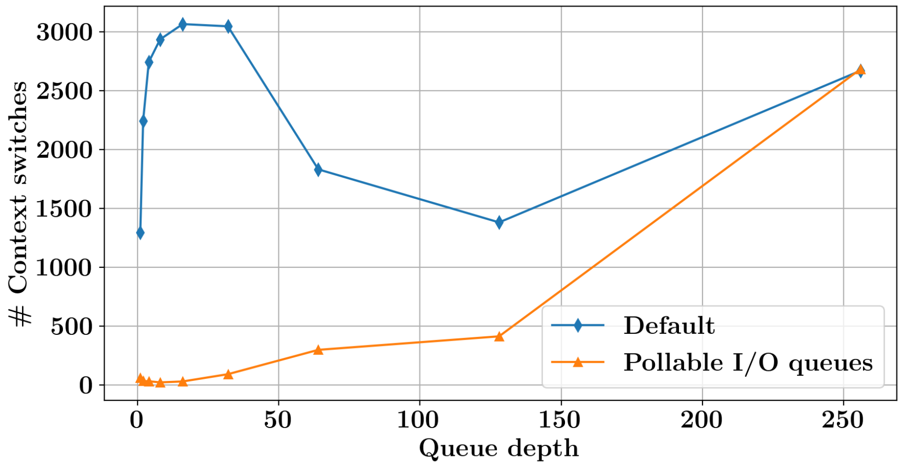

Figure 4.4: Plot for number of context switches vs Queue depth for default and pollable I/O queues configuration

3. This performance improvement is attributed to fewer context switches, as shown in Fig4.4, which depicts the number of context switches in the fio application over the experiment duration. Polling occurs within the same context as the application.

Therefore, enabling polling I/O queues in NVMeoF-TCP leads to improved performance in our setup.

Target idle polling

Next, we evaluate the performance of the idle target polling configuration. The target worker function is triggered whenever an NVMeoF-TCP request or NVMe response is enqueued into the corresponding NVMeoF-TCP target queue pair. However, there can be a delay between the enqueuing of a request/response and the start of the target worker function's execution. To mitigate this, NVMeoF-TCP offers an option to configure the target worker function to poll the queue pairs for a specified period (idle_poll_period_usecs).

When this polling period is specified, the worker function continuously checks the queue pairs for new requests or responses. If the queue is empty, the function requeues itself in the work queue until the elapsed time since the last execution with a request/response exceeds the configured polling period. This polling mechanism is expected to reduce the latency between the enqueueing of an item into the queue pair and the execution of the worker function.

To assess this, we need to evaluate two aspects: whether target polling improves performance, and what the optimal idle polling duration is. The effectiveness of the polling duration depends on the rate at which work is enqueued in the target queue pair, which is influenced by the rate of request arrival and request completion from the storage medium. Since these rates are system-specific, we will determine the polling duration that yields the best p99 latency in our setup for use in further experiments.

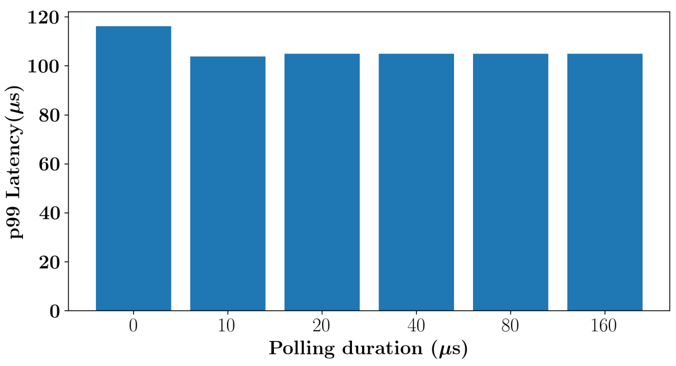

Figure 4.5: Plot for p99 latency(y-axis) for different target polling period(x-axis).

The following observations were made from the experiment results:

1. Fig. 4.5 shows that target idle polling reduces tail latency by 7.9-10.6% (9.21-12.29 μs). We select 40 μs as the target idle polling duration for better performance in our setup.

2. No improvement in throughput is observed for any polling duration. This is because, as the number of concurrent requests increases, the influx of requests and responses to the target queue also rises, keeping the worker function continuously active. As a

result, target idle polling does not contribute to performance gains in the throughput experiments since the worker function remains busy without the need for idle polling.

Thus, the finding from this section is that target idle polling improves tail latency.

4.3 Summary

In this chapter, we identified several configuration parameters and evaluated two configurations significantly impacting performance. The following observations were made during the evaluation:

- Pollable NVMeoF-TCP I/O queues: Improved p99 latency by 9.3% and increased single-core throughput by 14.4%.
- Idle target polling: Improved p99 latency by 7.9-10.6% depending on the polling duration but showed no improvement in IOPS.

Configuration	**Suggested value**
Number of Polling I/O queues	10 queue pairs (Number of cores in the initiator)
Idle target polling	40μs

Table 4.2: NVMeoF-TCP configuration used for upcoming experiments in this thesis

Based on the findings, we optimized our system configuration(configuration details are provided in Table4.2) for improved performance for the upcoming experiments addressing RQ2 and RQ3 to ensure better performance.

5

Evaluating Performance Overhead of NVMeoF-TCP over NVMe

In this chapter, we address **(RQ2) What is the performance overhead of NVMeoF-TCP compared to local NVMe?**. We evaluate the performance of the L-app, T-app, and B-app between NVMeoF-TCP and NVMe and quantify the performance overhead introduced by NVMeoF-TCP compared to direct NVMe access. The evaluation is carried out using the configuration settings identified from RQ1.

5.1 L-app Overhead Quantification

We begin by comparing the performance of L-app in NVMe and NVMeoF-TCP. Latency in NVMeoF-TCP can be attributed to the NVMeoF-TCP software stack, network transmission, and the storage medium, while in NVMe, it is influenced by the NVMe software stack and the storage medium. To isolate the overhead introduced by the NVMeoF-TCP software stack and network, we conduct two comparisons:

1. Software Overhead: We compare NVMe with NVMeoF-TCP on a localhost setup to identify the overhead introduced by the NVMeoF-TCP software stack.
2. Network Overhead: We compare NVMeoF-TCP on localhost and NVMeoF-TCP over 100 Gbps network setup to quantify the overhead caused by network transmission.

During the experiment for NVMeoF-TCP localhost, we observed that enabling completion polling in fio negatively impacted performance, as the polling interfered with NVMeoF-TCP's target request processing, both of which occurred on the same core. As a result, *we temporarily disabled completion polling for the NVMeoF-TCP localhost experiment only.*

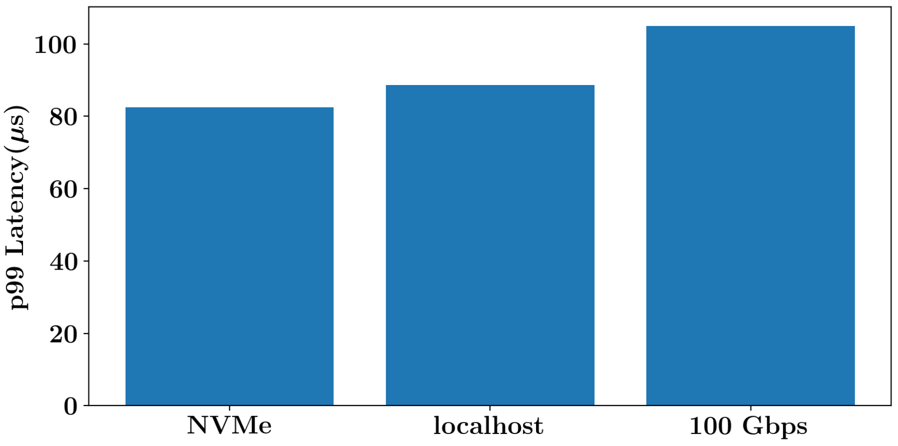

Figure 5.1: p99 latency plot of NVMe, NVMeoF-TCP(localhost) and NVMeoF-TCP(100 Gbps)

The results of the experiments are shown in Fig. 5.1, and the following observations can be made:

1. NVMeoF-TCP on localhost has 7.4% (6.14 μs) higher 99th percentile latency compared to NVMe.
2. NVMeoF-TCP over a 100 Gbps network shows a 19.6% (17.40 μs) increase in latency compared to NVMeoF-TCP on localhost.
3. NVMeoF-TCP over a 100 Gbps network exhibits a 27.3% (22.53 μs) increase in latency compared to NVMe.

These observations lead to the finding that in our setup, In NVMeoF-TCP over a 100 Gbps network, the network contributes more to the tail latency than the NVMeoF-TCP software stack.

5.2 T-app Overhead Quantification

Next, we compare the T-app performance in NVMe and NVMeoF-TCP over 100 Gbps, focusing on the maximum single-core throughput. To do this, we run a single fio process with varying queue depths until the throughput saturates. Note: Completion polling has been enabled. The experiment results are shown in Fig5.2.

The observations from the results are as follows:

1. For NVMe, a single core reaches saturation at a queue depth (QD) of 256, delivering 332.68 KIOPS.

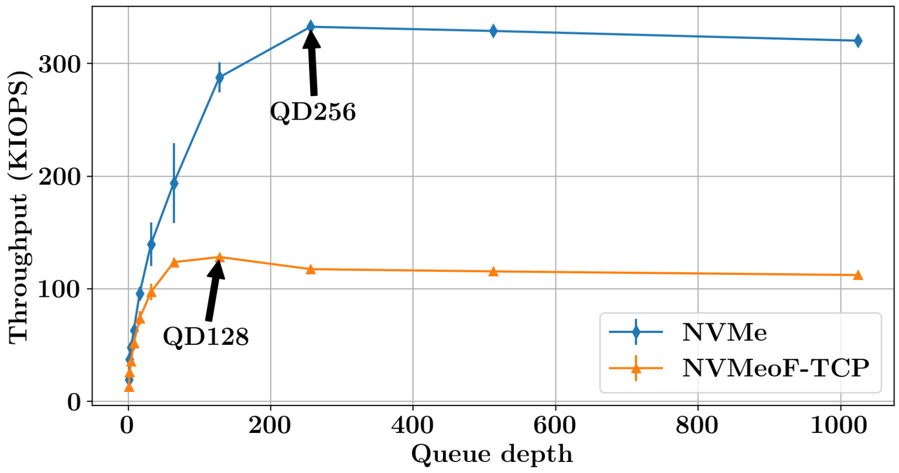

Figure 5.2: Throughput plot for NVMe and NVMeoF-TCP

2. For NVMeoF-TCP, a single core saturates at a QD of 128, providing around 128.24 KIOPS.

These observations indicate that, in our setup, a single core in the local NVMe stack provides 2.58 times the throughput of NVMeoF-TCP.

5.3 B-app Overhead Quantification

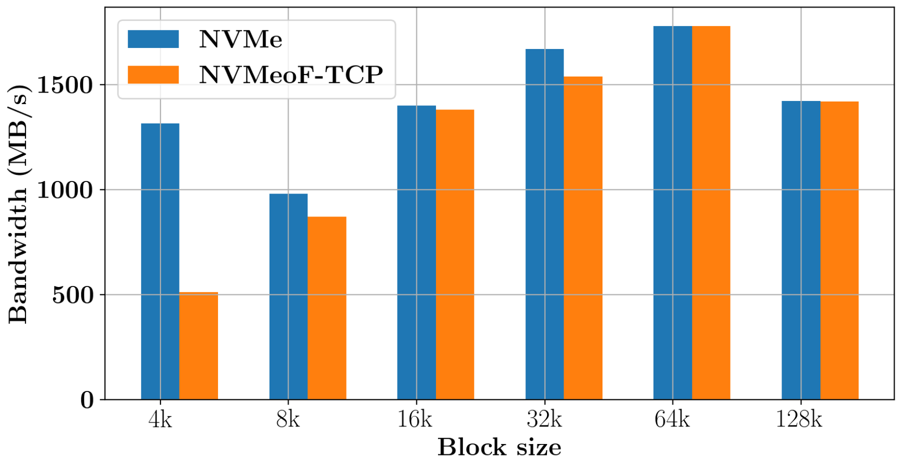

Figure 5.3: Bandwidth plot for NVMe and NVMeoF-TCP for different block sizes

Next, we compare the B-app performance in NVMe and NVMeoF-TCP. For this, we run a single fio process with varying queue depths for different block sizes until the bandwidth saturates. The result of the experiment is shown in Fig5.3 (contains only max achieved bandwidth for each block size). The observations from the plot are,

1. The performance gap between NVMe and NVMeoF-TCP is high at smaller block sizes.
2. As the block size increases, the performance difference narrows. At 64k, both NVMe and NVMeoF-TCP achieve similar bandwidth performance. This is because the SSD device in the setup was saturated.
3. There are two unexplained scenarios: The bandwidth drop at 128KB block size, where both NVMe and NVMeoF-TCP show a decline, is unclear. Also, the unexpectedly high bandwidth was observed with a 4KB block size in NVMe.

In our setup, both NVMe and NVMeoF-TCP reach the maximum bandwidth performance of the SSD when using a single core at larger block sizes.

5.4 Summary

In this section, we compared the performance of different applications(L-app, T-app, and B-app) in NVMe and NVMeoF-TCP and quantified the overhead of NVMeoF-TCP in our setup.

- L-app: NVMeoF-TCP exhibits higher latency than NVMe due to the overhead introduced by the software stack and network transmission. Specifically, NVMeoF-TCP on localhost shows a 7.4% higher p99 latency compared to NVMe, and over a 100 Gbps network, the latency increases by 27.3%. **The observation is that network transmission contributes more to tail latency than the NVMeoF-TCP software stack in our setup.**
- T-app: When **comparing single-core throughput in our setup, NVMe delivers 2.58 times the throughput of NVMeoF-TCP.** NVMe saturates at a queue depth of 256, achieving around 332.68 KIOPS, while NVMeoF-TCP reaches saturation at a queue depth of 128, providing only 128.24 KIOPS.
- B-app: **Both NVMe and NVMeoF-TCP achieve the maximum bandwidth performance of the SSD when using a single core at larger block sizes in our setup.**

6 Evaluating Performance Interference in NVMeoF-TCP

In this section, we address **(RQ3) What is the performance interference with various loads on compute resources in the NVMeoF-TCP stack?** We identify the different compute resources within the NVMeoF-TCP stack and determine their saturation points. After that, we outline the experimental design used to measure the interference for each compute resource. Finally, we perform interference characterization by conducting experiments for each resource.

6.1 Compute Resources and the Saturation Points in NVMeoF-TCP

Resource	**Saturation point (KIOPS)**	**Saturation configuration**
Initiator core	128.24	P1.QD128
Target core	180.30	P2.QD128

Table 6.1: Saturation points of compute resources in the NVMeoF-TCP stack and the corresponding fio configuration to obtain the saturation point.

In our setup, the compute resources in the NVMeoF-TCP stack include the initiator core and target core. Based on the saturation points detailed in Table 6.1, we observed that the initiator core reaches saturation before the target core in NVMeoF-TCP.

6.2 Experiment Design to Quantify Interference in NVMeoF-TCP

To characterize interference for a resource, we make the foreground flow from the L-app or T-app share the resource with a background flow under varying load levels. These loads range from 10%, 20% of the resource's saturation point up to the maximum load (Max), representing the highest load the background flow can exert while still allowing the foreground process to share that resource. For example, in the experiment of characterizing latency interference of the initiator core, we make the L-app fio process share the initiator core with the background fio process and note down the tail latency for the background app's various IOPS throttle limit of 25.64 KIOPS (0.20*128.24 KIOPS), 51.28 KIOPS (0.40*128.24 KIOPS) so on till the throughput of the background process is no longer increasing which we mark down as max load percentage of the background process with L-app sharing the initiator resources.

The experiments aim to answer the following questions about each resource:

1. How does varying load on the resource affect interference on the L-app?
2. How does varying load on the resource affect interference on the T-app?
3. What is the maximum background load that can be applied while the L-app shares the resource?
4. What is the maximum background load that can be applied while the T-app shares the resource?

The results from these questions will help characterize the interference behaviour of the resources.

Quantification of interference

For L-app, we quantify the interference using the formulae:

$$\text{Latency Interference} = \frac{\text{p99 Latency}<\text{concurrent load}>}{\text{p99 Latency}<\text{isolated}>}$$

The above formulae specify the factor of increase in the p99 latency of the L-app under load compared to the isolated situation. So, lower interference values for L-app are better, indicating less latency degradation under load.

For T-app, we quantify the interference using the formulae:

$$\text{Interference} = \frac{\text{Metric}<\text{concurrent load}>}{\text{Metric}<\text{isolated}>}$$

We call the interference metric of T-app as throughput sustained because the formula will have the value of 1 if there is no interference and approaches 0 as interference increases. So, in this case, the higher the "throughput sustained" value, the less interference there is with the T-app.

Fixing the Queue depth of T-app

For the interference study, We fix the queue depth of the T-app to 128. This decision is made to avoid conducting multiple interference studies for varying queue depths, as QD-128 represents all T-app scenarios. Specifically, QD-128 is chosen because it is the maximum queue depth at which a single fio process reaches saturation in the NVMeoF-TCP environment.

Network and other interference

Though the network is excluded from the interference study, both the network and the processors on the initiator and target are still shared between the foreground and background flows in all experiments due to limitations in the setup. As a result, interference may be present between the network and shared processors across the experiments. To account for this, a specific experiment was conducted where the network and processors were the only shared resources between the background and foreground flows, allowing for the isolation and measurement of their impact on performance. In this test, the background load was set to the saturation point of both the initiator and target cores to establish the maximum possible interference from the network and processors for the following experiments. The results indicated minimal interference, suggesting that the shared resources had a negligible effect on the performance in these experiments.

6.3 Interference Characterization of the Initiator

We start by examining the interference characteristics of the initiator core. In this experiment, we use the following configuration,

- 1 foreground Fio process (L-app/T-app) in core 0 of the initiator
- 1 background Fio process in core 0 of the initiator

- The foreground Fio process sends I/O requests to remote SSD1
- the background Fio process sends I/O requests to remote SSD2
- The TCP connection for queue pair in core 0 for remote SSD1, through which the foreground Fio process sends I/O requests, is mapped to core 0 in the target.
- The TCP connection for queue pair in core 0 for remote SSD2, through which the background Fio process sends I/O requests, is mapped to core 1 in the target.

The above setup ensures that the experiment measures interference explicitly caused by sharing the initiator core. Further sanity check was also performed, which checks the following invariants

- The Number of I/O requests completed by the foreground Fio process equals the number of NVMe requests in the core 0 of the target.
- The Number of I/O requests completed by the foreground Fio process equals the number of NVMe requests submitted to SSD1 in the target from any core.

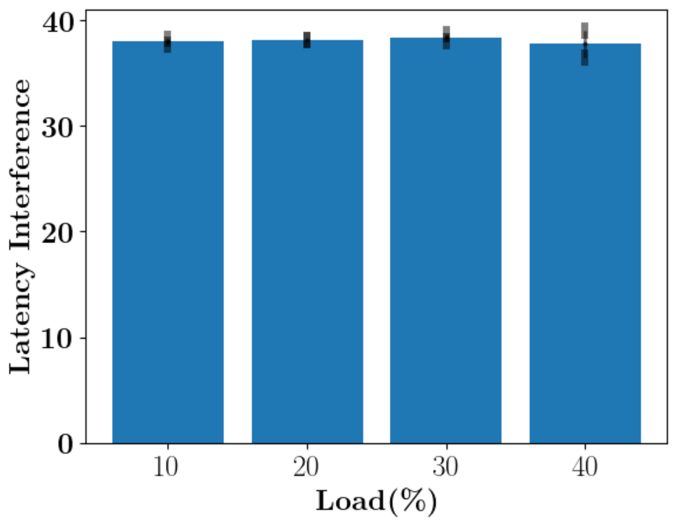

Figure 6.1: Plot for latency interference (y-axis) for various background loads (x-axis) in the initiator core. The higher the y-axis, the higher the latency interference.

The plots in Fig 6.1 illustrate the interference to the tail latency of the L-app under various background loads. From the plot, the following observations can be made:

1. The latency interference remains consistently around 38x, regardless of the background load percentage. This high level of interference, irrespective of load variation, may result from the polling by the Fio process.

2. The max achieved background flow load with an L-app is 40% of the saturation point of the initiator.

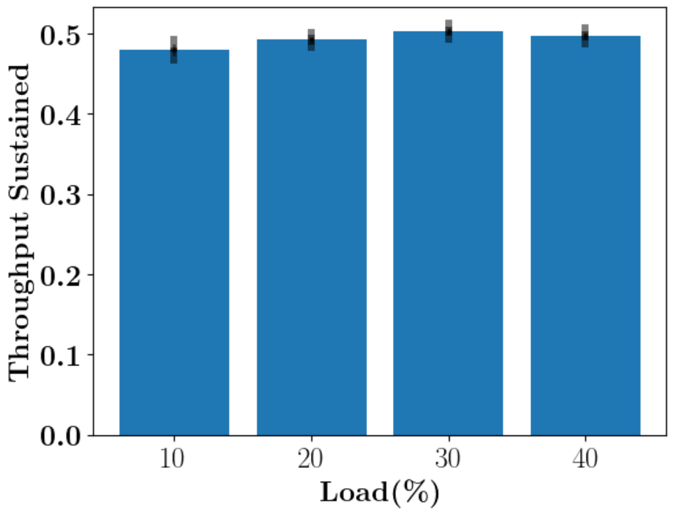

Figure 6.2: Plot for throughput interference (y-axis) for various background loads (x-axis) in the initiator core. The lower the y-axis, the higher the throughput interference.

The plots in Fig. 6.2 show the interference on the throughput of the T-app under various background loads. From the plot, we can observe the following:

1. The throughput of T-app reduces to 49% irrespective of the background load. This high level of interference, irrespective of load variation, may result from the polling by the Fio process.

2. The maximum background load achieved with the T-app is 40%.

6.4 Interference Characterization of the Target

Next, we examine the interference characteristics of the target core. In this experiment, we use the following configuration,

- 1 Foreground Fio process (L-app/T-app) in core 0 of the initiator
- 2 Background Fio processes in cores 1 and 2 of the initiator, respectively. We are using two processes because they are required to saturate the target resource.
- The foreground Fio process sends I/O requests to remote SSD1
- the background Fio process sends I/O requests to remote SSD2

- The TCP connection for queue pair in core 0 for remote SSD1, through which the foreground Fio process sends I/O requests, is mapped to core 0 in the target.
- The TCP connections for core 1 and 2 queue pairs for remote SSD2, through which the background Fio processes send I/O requests, are mapped to core 0 in the target.

The above setup isolates and observes interference caused by sharing the target core. Further sanity check was also performed, which checks the following invariants

- The Number of I/O requests completed by the foreground Fio process equals the number of NVMe requests in the core 0 of the target for SSD1.
- The Number of NVMe requests submitted from other cores expect core 0 is 0 in the target.

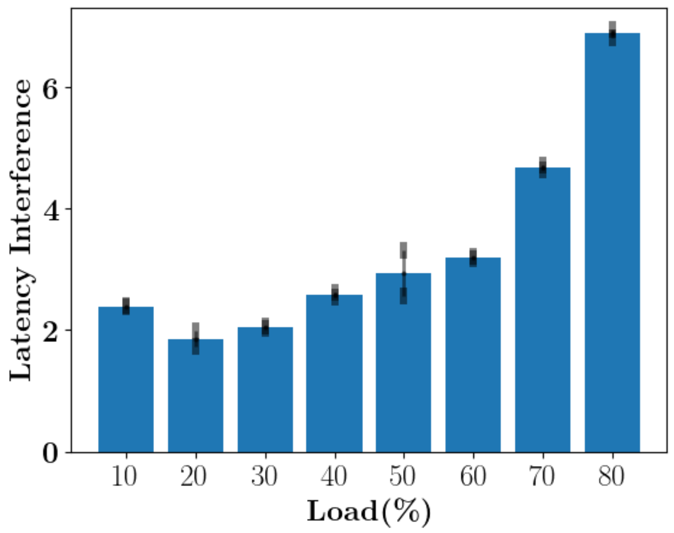

Figure 6.3: Plot for latency interference (y-axis) for various background loads (x-axis) in the target core. The higher the y-axis, the higher the latency interference.

The plot in Fig6.3 shows the tail latency interference of the L-app for various background loads. The following can be observed from the plot,

1. Interference increases as the load on the target core rises due to background flows. Between 20% and 60% load, the interference increases linearly from 1.8x to 3.2x. However, from 70% to 80% load, there is a significant jump to 6.8x interference.

The plot in Fig6.4 shows the throughput interference of the T-app for various background loads. The following can be observed from the plot,

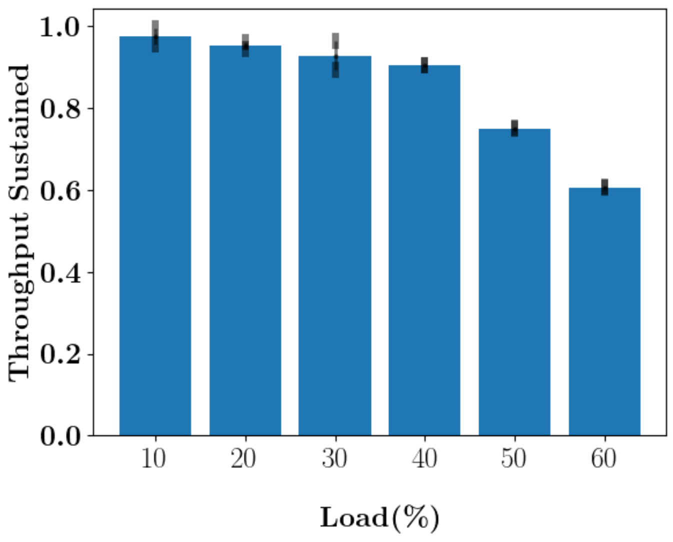

Figure 6.4: Plot for throughput interference (y-axis) for various background loads (x-axis) in the target core. The lower the y-axis, the higher the throughput interference.

1. Interference increases as the background load on the target core rises. At a 10% load on the target core, there is no noticeable impact on the T-app's performance. However, as the load increases to 20%-40%, T-app's throughput decreases by 5%-10%. At a 50% load, the throughput reduction reaches 25%, and at 60% load, T-app experiences a significant 40% drop in throughput.

6.5 Summary

In this section, we quantified the interference of the initiator and target compute resources in the NVMeoF-TCP stack. The analysis shows different patterns of performance interference for initiator and target components when background loads change.

For the initiator, latency interference stays consistently high at around 38x, regardless of the background load. T-app's throughput on the initiator also drops by 49%, and this reduction remains the same no matter how the background load changes.

For the target, interference increases as the target core load increases. Between 20% and 60% load, interference grows steadily from 1.8x to 3.2x. However, there is a sharp rise at higher loads, jumping to 6.8x between 70% and 80%. In terms of throughput, there is no significant effect at a 10% load, but it starts to drop by 5%-10% as the load increases to 20%-40%. At a 50% load, throughput falls by 25%, and at 60% load, it drops significantly by 40%.

7 Discussion

7.1 Limitations of the Study

- **No Evaluation of Write Performance**: This study only focuses on read performance, and we have not evaluated how write operations perform. Since write tasks behave differently from reads regarding speed and resource use, this leaves a gap in understanding how the system handles write-heavy workloads or mixed read-write environments.

- **Lack of Network Interference Study**: We did not analyze network interference in detail, even though the network is a shared resource. Without this, we might miss out on how network congestion or traffic impacts the overall performance of NVMeoF-TCP.

- **No Analysis of Configuration Parameters Impact on Interference**: We have not explored how our specific configuration parameters (such as pollable I/O queues and target polling) affect resource interference. This limits our understanding of whether these configurations help mitigate or worsen interference under different load conditions.

Addressing these limitations in future research would provide a more comprehensive understanding of NVMeoF-TCP performance and interference characterization.

7.2 Related work

Performance Characterization

Khosravi et al.(16) conducted a detailed performance analysis of iSCSI, characterizing the CPU and memory requirements for various I/O sizes and providing a breakdown of the processing costs at different levels of the iSCSI stack. Their work offers insights into the resource demands associated with iSCSI-based storage solutions. Kilimovic et al.(17) focused on tuning remote access to Flash over iSCSI and evaluated its impact on workloads sampled from real datacenter applications. Their analysis showed that while remote Flash access via iSCSI leads to a 20% drop in throughput at the application level, disaggregation helps to mitigate these overheads by enabling resource-efficient scale-out, making it possible to balance the performance losses with enhanced scalability.

Guz et al.(18) conduct a comprehensive performance characterization of NVMeoF using RDMA as a transport protocol, comparing it with iSCSI and Direct-Attached Storage (DAS). The study examines performance metrics such as latency and throughput, highlighting how NVMeoF-RDMA provides superior performance over traditional storage technologies like iSCSI. The authors also explore the impact of using SPDK (Storage Performance Development Kit) on the NVMeoF-RDMA target module, showing how SPDK optimizes I/O paths to further enhance performance. Similarly, Kashyap et al.(19) extend this analysis by comparing the performance of different NVMeoF transport protocols using SPDK on the target side.

Xu et al.(20) analyzed the performance of NVMeoF using both TCP and RDMA, with and without target offloading. Their findings revealed that offloading reduced CPU utilization by 38.7% and improved overall performance, underscoring the potential benefits of offloading mechanisms in optimizing NVMeoF deployments.

Flash Storage Interference Characterization

Ren et al.(21) examine the performance overheads and interference effects introduced by different Linux I/O schedulers in local NVMe-based storage, focusing on the Kyber scheduler. By analyzing Kyber's configuration space, they offer recommendations on how to minimize interference in storage systems. In another study, Doekemeijer et al.(22) focus on Zoned Namespace (ZNS) devices, characterizing the interference between I/O operations and I/O management tasks. Their findings led to the development of the ConfZNS++ emulator, which improves the handling of I/O operations in the ConfZNS emulator(23).

Multiple Resources Scheduling

Gimbal(24) is a software storage switch that efficiently manages I/O traffic between Ethernet ports and NVMe drives for co-located tenants. It implements a delay-based SSD congestion control algorithm that helps estimate the cost of operations, allowing the system to create virtual slots for I/O tasks dynamically. These virtual slots are then divided among users, ensuring fair and efficient resource distribution. Ng et al.(25) introduce a protocol called NVMe-over-Priority-Fabrics (NVMe-oPF), which supports multi-tenancy and allows applications to specify whether to optimize for latency or throughput. On the other hand, Gupta et al.(26) propose modifying congestion control mechanisms to prioritize specific I/O flows, allowing applications with low latency or high throughput storage requirements to receive enhanced network support. RackBlox,(27) introduces a system where flash storage devices are directly connected to network switches, which handle I/O scheduling. This scheduling is based on both the priority of the task's completion and the state of the flash storage replicas.

8

Conclusion

In this thesis, we conducted a performance characterization of NVMeoF-TCP, analyzing both its overhead compared to NVMe and the interference it causes in various resources. We began by examining the available configuration options for NVMeoF-TCP and their effects on performance. Next, we quantified the overhead introduced by NVMeoF-TCP when compared to direct NVMe access. Finally, we performed an interference characterization for the initiator and target resources in NVMeoF-TCP environments.

In this section, we summarize the answers to the research questions presented at the start of this thesis, and we also discuss potential areas for future research.

8.1 Research Question

(RQ1) What are the performance implications of NVMeoF-TCP polling configurations?

In Chapter 4, we identified the polling configuration options of NVMeoF-TCP: Polling I/O queues and Target idle polling. Our evaluation showed that enabling polling I/O queues improves tail latency by 9% and increases single-core throughput by 14%. Similarly, enabling target idle polling reduced tail latency by 8-10%, depending on the polling duration. These results indicate that enabling polling configurations can significantly enhance the performance of NVMeoF-TCP.

(RQ2) What is the performance overhead of NVMeoF-TCP compared to local NVMe?

In Chapter 5, we quantified the overhead of NVMeoF-TCP compared to NVMe. We found that the NVMeoF-TCP software stack introduces a 7% performance overhead compared to local NVMe, while NVMeoF-TCP over a 100 Gbps network increases this overhead to 28%. In terms of throughput, NVMeoF-TCP achieves only 40% of the throughput that NVMe can deliver from a single core. However, NVMe and NVMeoF-TCP saturate the SSD in our setup at higher block sizes. These results state the need to account for additional overheads when deploying NVMeoF-TCP, particularly in high-performance environments.

(RQ3) What is the performance interference with various loads on compute resources in the NVMeoF-TCP stack?

In Chapter 6, we explored the interference characteristics of NVMeoF-TCP by analyzing how different compute resources, such as the initiator core and target core, are impacted under varying background loads. We found that the initiator core consistently showed high latency interference (around 38x) and a 49% reduction in T-app throughput, unaffected by load variation. In contrast, the target core's interference increased steadily with background load, rising from 1.8x at 20% load to 3.2x at 60%, with a sharp jump to 6.8x at 80%. T-app throughput on the target dropped significantly at higher loads, with up to a 40% decrease at 60% load. These findings highlight the need for specific load management strategies across different resources to minimize performance interference in NVMeoF-TCP systems.

8.2 Guidelines

Based on the findings from the research questions (RQ1, RQ2, and RQ3), the following guidelines can help improve performance and manage resources in NVMeoF-TCP environments:

Configuring for Better Performance

- **Enable Pollable I/O Queues:** Configuring pollable I/O queues can significantly reduce latency and improve throughput.

- **Adjust Target Idle Polling Duration:** Configuring target idle polling can lead to latency improvements. However, the optimal polling duration depends on the incoming request rate.

Account for Performance Overhead from Disaggregation

- **Higher Latency and Lower Throughput:** Compared to local NVMe, NVMeoF-TCP introduces a 27.3% increase in latency and a reduction in single-core throughput by 2.58 times. When deploying NVMeoF-TCP, plan for these overheads.
- **Larger Block Size for Bandwidth-Sensitive Applications:** Applications that rely on high bandwidth in NVMeoF-TCP storage setups should use larger block sizes to mitigate the performance effects of disaggregation. Larger block sizes help maintain bandwidth levels similar to NVMe's local storage.

Minimize Resource Interference

- **Balance Load Distribution Across Initiator:** The initiator experiences consistent latency interference, regardless of background load. To avoid bottlenecks, distribute tasks and background processes across multiple cores or servers to alleviate pressure on a single initiator.
- **Monitor and Limit Background Loads on Target Cores:** Interference increases with higher background loads on target cores. To prevent significant drops in performance, keep the load on target cores below 70% for L-apps and 50% for T-apps.

8.3 Future Work

This study focused only on NVMeoF-TCP. Future research could examine NVMeoF-RDMA to understand how it performs and handles interference in cloud environments compared to NVMeoF-TCP.

Additionally, this study mainly examined interference in the initiator and target resources. Future work should include other vital resources like the network and storage to provide a more complete understanding of interference. This would give a more comprehensive picture of how interference affects performance in the NVMeoF-TCP stack.

Another area for future research is the development of a multi-resource scheduler. This scheduler would help manage resources like CPU, memory, network, and storage more efficiently, ensuring consistent Quality of Service (QoS) across all tenants by improving resource allocation and reducing interference.

References

[1] Alexandru Iosup, Fernando Kuipers, Ana Lucia Varbanescu, Paola Grosso, Animesh Trivedi, Jan Rellermeyer, Lin Wang, Alexandru Uta, and Francesco Regazzoni. **Future Computer Systems and Networking Research in the Netherlands: A Manifesto**, 2022. 1

[2] Hystax. **Cloud Infrastructure: The Backbone of Modern Computing**, 2023. Accessed: 2024-10-09. 1

[3] Deloitte. **Milliseconds Make Millions: Why Speed Matters in Financial Services**, 2021. Accessed: 2024-09-12. 1

[4] Isaac Odun-Ayo, Sanjay Misra, Olusola Abayomi-Alli, and Olasupo Ajayi. **Cloud Multi-Tenancy: Issues and Developments**. In *Companion Proceedings of The10th International Conference on Utility and Cloud Computing*, UCC '17 Companion, page 209–214, New York, NY, USA, 2017. Association for Computing Machinery. 1

[5] Zhen Cao, Vasily Tarasov, Hari Prasath Raman, Dean Hildebrand, and Erez Zadok. **On the performance variation in modern storage stacks**. In *15th USENIX conference on file and storage technologies (FAST 17)*, pages 329–344, 2017. 1

[6] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. 2013. 1

[7] James Hamilton. **Internet-scale service infrastructure efficiency**. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, page 232, New York, NY, USA, 2009. Association for Computing Machinery. 1

[8] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. **Network support for resource disaggregation in next-generation datacenters**. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, New York, NY, USA, 2013. Association for Computing Machinery. 1

[9] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. **An analysis of performance evolution of Linux's core operations**. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 554–569, New York, NY, USA, 2019. Association for Computing Machinery. 3

[10] Raj Jain. *The art of computer systems performance analysis*. john wiley & sons, 1990. 3

[11] Sonja Bezjak, April Clyburne-Sherin, Philipp Conzett, Pedro L Fernandes, Edit Görögh, Kerstin Helbig, Bianca Kramer, Ignasi Labastida, Kyle Niemeyer, Fotis Psomopoulos, et al. **The open science training handbook**. 2018. 3

[12] Lorrie Cranor, Kim Hazelwood, Daniel Lopresti, and Amanda Stent. **Conference Submission and Review Policies to Foster Responsible Computing Research**. *arXiv preprint arXiv:2408.09678*, 2024. 3

[13] Blesson Varghese and Rajkumar Buyya. **Energy-efficiency and sustainability in new generation cloud computing: A vision and directions for integrated management of data centre resources and workloads**. *Future Generation Computer Systems*, **82**:128–137, 2018. 4

[14] Keith Busch et al. **NVMeCLI - NVMe Command Line Interface**. `https://github.com/linux-nvme/nvme-cli`, 2016. Accessed: 2024-10-11. 9

[15] SPDK Project. **SPDK NVMe-oF TCP (Target & Initiator) Performance Report**. Technical report, Storage Performance Development Kit (SPDK), 2021. Accessed: 2024-10-20. 16

[16] H.M. Khosravi, Abhijeet Joglekar, and Ravi Iyer. **Performance characterization of iSCSI processing in a server platform**. In *PCCC 2005. 24th IEEE International Performance, Computing, and Communications Conference, 2005.*, pages 99–107, 2005. 38

[17] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. **Flash storage disaggregation**. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery. 38

[18] Zvika Guz, Harry Li, Anahita Shayesteh, and Vijay Balakrishnan. **Performance characterization of nvme-over-fabrics storage disaggregation**. *ACM Transactions on Storage (TOS)*, **14**(4):1–18, 2018. 38

[19] Arjun Kashyap, Shashank Gugnani, and Xiaoyi Lu. **Impact of commodity networks on storage disaggregation with nvme-of**. In *Benchmarking, Measuring, and Optimizing: Third BenchCouncil International Symposium, Bench 2020, Virtual Event, November 15–16, 2020, Revised Selected Papers 3*, pages 41–56. Springer, 2021. 38

[20] Jiexiong Xu, Yue Qiu, Yiquan Chen, Yijing Wang, Wenhai Lin, Yiquan Lin, Shushu Zhao, Yuqi Liu, Ying Wang, and Wenzhi Chen. **Performance Characterization of SmartNIC NVMe-over-Fabrics Target Offloading**. In *Proceedings of the 17th ACM International Systems and Storage Conference*, SYSTOR '24, page 14–24, New York, NY, USA, 2024. Association for Computing Machinery. 38

[21] Zebin Ren, Krijn Doekemeijer, Nick Tehrany, and Animesh Trivedi. **BFQ, Multiqueue-Deadline, or Kyber? Performance Characterization of Linux Storage Schedulers in the NVMe Era**. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24, page 154–165, New York, NY, USA, 2024. Association for Computing Machinery. 38

[22] Krijn Doekemeijer, Dennis Maisenbacher, Zebin Ren, Nick Tehrany, Matias Bjørling, and Animesh Trivedi. **Exploring I/O Management Performance in ZNS with ConfZNS++**. In *Proceedings of the 17th ACM International Systems and Storage Conference*, SYSTOR '24, page 162–177, New York, NY, USA, 2024. Association for Computing Machinery. 38

[23] Inho Song, Myounghoon Oh, Bryan Suk Joon Kim, Seehwan Yoo, Jaedong Lee, and Jongmoo Choi. **ConfZNS: A Novel Emulator for Exploring**

Design Space of ZNS SSDs. In *Proceedings of the 16th ACM International Conference on Systems and Storage*, SYSTOR '23, page 71–82, New York, NY, USA, 2023. Association for Computing Machinery. 38

[24] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. **Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs**. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 106–122, New York, NY, USA, 2021. Association for Computing Machinery. 39

[25] Darren Ng, Andrew Lin, Arjun Kashyap, Guanpeng Li, and Xiaoyi Lu. **NVMe-oPF: Designing Efficient Priority Schemes for NVMe-over-Fabrics with Multi-Tenancy Support**. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 519–531, 2024. 39

[26] Jit Gupta, Krishna Kant, Amitangshu Pal, and Joyanta Biswas. **Configuring and Coordinating End-to-end QoS for Emerging Storage Infrastructure**. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, **9**(1):1–32, 2024. 39

[27] Benjamin Reidys, Yuqi Xue, Daixuan Li, Bharat Sukhwani, Wen-Mei Hwu, Deming Chen, Sameh Asaad, and Jian Huang. **RackBlox: A Software-Defined Rack-Scale Storage System with Network-Storage Co-Design**. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 182–199, 2023. 39

9

Appendix

9.1 Artifact Description

9.1.1 Abstract

This artifact description describes how to set up the benchmarking environment and reproduce the results as seen in the thesis.

9.1.2 Artifact Checklist

- Program: The benchmarking scripts are available at https://github.com/bergstartup/NVMeoFTCP-Characterization.
- Publicly available?: The scripts are publicly available
- Code license: GPL-3.0 license

9.1.3 Description

How to access

All the scripts necessary to run benchmarks to obtain the graphs in this thesis can be obtained from GitHub with:

```
$ git clone https://github.com/bergstartup/NVMeoFTCP-Characterization
```

Software Dependencies

The following software are needed to run the benchmarks and to make the plots:

- Fio-3.37

- bpftrace-0.20.2
- Python 3

Software and Hardware Configuration

All benchmarks run on top of Qemu 6.1.0 with KVM enabled
Hardware configuration host (Initiator and Target)

- 20-core 2.40GHz Intel(R) Xeon(R) Silver 4210R CPU with two sockets connected in NUMA mode. Each socket has ten physical cores and one thread for each core.
- 252GB of DDR4 RAM

Hardware configuration VM (Initiator)

- 10-core 2.40GHz Intel(R) Xeon(R) Silver 4210R
- 24GB of DDR4 RAM
- Passthrough of Mellanox ConnectX-5 NIC

Hardware configuration VM (Target)

- 10-core 2.40GHz Intel(R) Xeon(R) Silver 4210R
- 24GB of DDR4 RAM
- Passthrough of Mellanox ConnectX-5 NIC
- Passthrough of 2X Western Digital SN540 SSD

The initiator and the target should have a 100Gbps link connected through the Mellanox NIC. The QEMU VM process, memory and all peripherals should be pinned to the same NUMA domain in the host, both the initiator and target. The Operating System used in the VM is Ubuntu 24.04 with Linux Kernel 6.8.0.

Experiment workflow

Setup initialization

In target

```
$ cd NVMeoFTCP-Characterization
$ cd ./scripts/setup
$ ./nvmeof_tcp_target_setup.sh $dev_path_0 0
$ ./nvmeof_tcp_target_setup.sh $dev_path_1 1
```

NVMeoF-TCP configuration experiments (RQ1)

Polling I/O queues

```
$ cd NVMeoFTCP-Characterization
$ cd ./scripts/benchmark/fio/performance
$ ./poll_npoll.sh
/* After execution of tests */
$ python3 crunching.py
```

Target Idle Polling

```
$ cd NVMeoFTCP-Characterization
$ cd ./scripts/benchmark/fio/performance
$ ./tpoll.sh
/* After execution of tests */
$ python3 crunching.py
```

NVMeoF-TCP overhead experiments (RQ2)

Latency overhead (In the target)

```
$ cd NVMeoFTCP-Characterization
$ cd ./scripts/benchmark/fio/performance
$ ./local_bench.sh
$ ./local_tcp.sh
```

Latency, Throughput and Bandwidth Overhead (In the initiator)

```
$ cd NVMeoFTCP-Characterization
$ cd ./scripts/benchmark/fio/performance
$ ./remote_bench.sh
$ ./copy_local.sh
$ python3 crunching.py
```

NVMeoF-TCP interference experiments (RQ3)

For initiator interference

```
$ cd NVMeoFTCP-Characterization
$ cd ./scripts/benchmark/fio/qos
$ ./initiator.sh
$ python3 crunching.py
```

For target interference

```
$ cd NVMeoFTCP-Characterization
$ cd ./scripts/benchmark/fio/qos_ns
$ ./target.sh
$ python3 crunching.py
```