NeSt: A QoS Differentiating End-to-End Networked Storage Simulator

Jit Gupta[a], Sourav Das[a], Krishna Kant[1a]

[a] *Computer and Information Sciences, Temple University, 1925 N 12th St, Philadelphia, 19122, PA, USA*

Abstract

The emerging high-speed storage technologies increasingly use Nonvolatile Memory Express (NVMe) protocol to meet their high throughput and low latency needs. In a datacenter environment, applications accessing multiple such devices over the fabric (i.e. the network) tend to have Quality of Service (QoS) requirements pertaining to offered throughput and experienced latency. In this paper we describe a networked storage system simulator called NeSt that supports end-to-end (E2E) QoS differentiation across multiple classes of service. This is done by conveying the class designation end to end and using it to consistently but independently apply the differentiation in each segment of the path. We demonstrate the ability of NeSt to provide end-to-end QoS differentiation under a variety of situations. To the best of our knowledge, NeSt is the first simulator of networked storage (consisting of multiple NVMe SSDs) that supports E2E QoS differentiation.

Keywords: Networked Storage; End to End QoS differentiation; NVMe Protocol; Congestion Management;

1. Introduction

Storage access is a fundamental service in data centers and is routinely provided by specially designed storage servers that host multiple storage devices, manage the overall storage capacity, and provide storage access to compute servers throughout the data center. Thus the network is an integral part of such a storage system and is easily congested by the emerging storage

[1] This research was supported by NSF grant CNS-2011252 and an Intel grant

technologies that can drive multi-gigabyte per second IO rate per storage device. The networked storage architecture is necessitated by many factors including the need to allocate arbitrary amounts of storage for applications regardless of the capacities of the devices inside the storage server, and the complex storage management that storage servers enable. A storage server may contain storage devices of varying characteristics including the emerging persistent memory, flash technology-based Solid State Disks (SSDs), and traditional magnetic hard drives (HDDs). Since SSDs have largely replaced HDDs for primary storage, we consider SSD-based storage and persistent memory only for the networked storage simulator discussed here.

With applications becoming increasingly data-intensive [1], both storage bandwidth and storage access latency have become key drivers of application performance. In particular, congestion anywhere in the storage path can adversely affect the ability of the application to read the required data for processing or write the data that it is generating. Since different applications have varying levels of storage bandwidth needs and latency tolerance, QoS differentiation between different classes of applications becomes essential. It is intuitively clear, and demonstrated later, that all components of the storage path must work in unison to provide the required differentiation among various classes of traffic, or else the results could be unpredictable. The components of the end-to-end (E2E) path include the host storage stack, network, storage access protocol, and the storage device itself. These components come into play both for the request and the response. That is, the E2E path should consider both the forward (request) and backward (response) segments, and treat them consistently in terms of QoS. The simulator, called NeSt (for Networked Storage) discussed here, enables such a treatment using some built-in mechanisms, which the researchers can experiment with for different traffic mixes and relative treatment requirements. We plan to distribute this tool with GPL license, which allows the researchers to further enhance the implemented mechanisms or implement new ones and thereby help create a powerful tool for the storage community.

The key contribution of this paper is to provide a comprehensive simulator for networked storage that supports end-to-end QoS differentiation to enable storage systems research. To the best of our knowledge, such a simulator currently does not exist but is essential to study storage systems performance for emerging data-intensive applications that may demand high throughput and/or very low latency.

The rest of our paper is organized as follows - Section 2 talks about the background and motivation of this work along with the limitations of existing works. Section 3 discusses our proposed simulator - NeSt along with its features and the implementation efforts we undertook. Section 4 compares NeSt to existing open-sourced simulation tools while Section 5 evaluates NeSt. Finally, the paper is concluded in 6.

2. Background and Related Work

2.1. Storage End QoS

Modern SSD technology has made significant strides compared to traditional magnetic disks or HDDs, excelling in both access latency and data transfer rates. For example, even budget-friendly SSDs can now achieve impressive transfer rates of 25-35 Gb/sec and maintain response times under 100 microseconds [2, 3]. Newer storage technologies, such as Intel's Optane and Kioxia's XL-flash, take latency even lower, reaching around 10-20 microseconds [4]. Furthermore, both the capacities of individual devices and the storage requirements of applications continue to go up.

These advancements have substantial implications for how we organize and access storage devices. First, it should be possible to seamlessly allocate storage to applications across one or more devices. Second, the storage access protocols must accommodate high data transfer rates while keeping latency minimal so that protocol delays don't significantly impact overall access time. Third, since a few storage devices can easily congest a 100 Gb/sec link, the network latency becomes an important component of overall latency, especially under congestion. This means that the differentiation between applications based on their latency or throughput requirements becomes essential.

The NVMe protocol has risen as the dominant choice due to its ability to deliver both high throughput and low latency [5]. It achieves this through enhancements in the queuing process atop the PCIe interface, enabling devices to interact directly with the CPU, in contrast to older protocols like SATA and SAS that use indirect DMA interfaces. NVMe supports multiple "queue pairs" with varying priorities, allowing for a flexible approach to handling storage access. Extending storage access over a network necessitates the local access protocol, such as NVMe, to be transported from the host to the target without altering the fundamental access principles. NVMe over Fabric (NVMe-oF) [6] achieves this by encapsulating submission and completion queue entries into transport-independent "capsules" for transfer between the host and the target. This transport approach also stores the command queue in the device controller's memory, known as the Controller Memory Buffer (CMB), rather than in the host's memory, further reducing latency. The completion queue remains with the host [7].

NVMe-oF is designed to run over a reliable transport, and suitable options for data center environments include DCTCP (based on TCP) and DCQCN (based on RDMA). These transport protocols aim to eliminate packet losses and minimize latency, with DCQCN potentially offering lower latency. However, neither of them provides differentiation in quality of service (QoS) [8].

2.2. Storage vs. Persistent Memory

The emerging Persistent Memory (PM) technology offers low-latency, high-throughput, overwrite capability (unlike regular NAND flash), and very high endurance. These attributes makes it a viable option for direct DRAM-like access while providing data persistence. However, to take advantage of the persistence, it is necessary to deploy mechanisms such as transactional memory to ensure that the data consistency can be preserved following a crash or power failure.

While PM technologies, such as Intel Optane are currently significantly slower than the venerable Dynamic Random-Access Memory (DRAM), they are cheaper than DRAM and thus can be used to extend DRAM [9]. Leading tech giants like Amazon and Oracle have recognized the potential of PM and incorporated it into their offerings. For instance, Amazon ElastiCache[2] and Oracle's Persistent Memory Database leverage Persistent Memory (pmem) technology for data storage and database solutions.[3] PM devices can serve dual purposes, operating with either storage or memory semantics. Under storage semantics, access sizes are typically larger (e.g., 4KB), and threads requesting device access may be temporarily switched out until the desired data is received. Conversely, in the memory model, access sizes typically involve fetching a few cache lines, causing the CPU to stall until the data arrives. In this work, we consider the storage model pertaining to PM devices. Our previous work [10] has explored the use of such memory model.

[2]https://aws.amazon.com/elasticache/

[3]Intel has recently discontinued development of Optane due to business reasons, but current products remain available. See https://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory/optane-dc-persistent-memory.html.

While locally installed PM often benefits from a DRAM-like interface (e.g., DDR) to provide low latency, the scenario changes when accessing it over a network. In either case (whether being accessed as a storage device or a memory device), the additional network latency can be excessive, particularly when this traffic competes with other classes of traffic in the network. It is thus important to provide the highest priority to PM transfers so that the PM traffic is not throttled even using congestion episodes (assuming that it is relatively small as compared to the other non-PM traffic). Furthermore, the network switches should also be equipped to provide preference to this traffic, ensuring that it receives the highest priority end to end. It is thus possible to reduce remote PM access latency substantially.

2.3. QoS Limitations in Current Data Center Networks

Current data center networks struggle to deliver consistent Quality of Service (QoS) for diverse traffic types, particularly those with strict latency requirements. This challenge stems from several factors, including: the dominance of kernel-based network stacks, which can introduce bottlenecks and limit responsiveness; the lack of flexibility in handling different traffic classes with varying bandwidth and latency needs; and the difficulty in ensuring predictable performance across a complex network infrastructure [11]. As a result, applications with stringent QoS requirements, such as real-time analytics and high-frequency trading, often suffer from unpredictable delays and performance fluctuations. In order to mitigate latency issues and designate traffic based on different classes within the network, two types of scheduling mechanisms can be utilized: Relative Priority and Strict Priority.

Relative Priority: For flows where absolute timing isn't very important, relative priority shines. This allows fine-grained QoS differentiation based on bandwidth and latency needs without completely starving lower-priority flows. This aspect involves reducing the latency for traffic belonging to a high-priority QoS class compared to other classes during standard traffic conditions. NeSt supports various mechanisms for relative prioritization, including Weighted Round Robin (WRR). With WRR, each flow receives service in proportion to its assigned weight, ensuring fair and predictable resource allocation within the designated class. Other mechanisms, like QoS-aware versions of TCP and RDMA (QTCP and QRDMA), further refine prioritization based on network conditions and application requirements. We delve deeper into these advanced mechanisms in Section 2.4, exploring their nuanced control over QoS.

Strict Priority: For applications demanding guaranteed low latency, strict priority makes more sense. This is particularly relevant for workloads accessing the Persistent Memory Region (PMR) introduced by the NVMe specification [12]. PMR offers memory-level data access speeds, crucial for latency-sensitive applications. However, strict priority is not without its drawbacks. Throttling other flows during congestion can be undesirable, potentially leading to starvation for lower-priority traffic. This highlights the need for a more flexible approach for non-latency-critical workloads.

In the context of Ethernet and IP-based networks, Quality of Service (QoS) has been a subject of extensive study. Standardized mechanisms like Differentiated Services Code Point (DSCP) have been widely adopted, particularly at the IP layer [13]. However, DSCP was primarily designed for wide area networks, where transport protocols, such as conventional TCP, are sensitive to packet losses, making a packet loss-centric QoS treatment reasonable. In contrast, high-speed data center networks try to avoid packet losses entirely because of their very detrimental impact. This renders DSCP less suitable. Moreover, DSCP operates on a "hop-by-hop" basis within each router, rather than providing uniform end-to-end treatment in layer-2 switch-centric networks in data centers. Several lossless transports have been proposed in the literature for data centers, including Data Center TCP (DCTCP) on the TCP side and Data Center Quality of Service Congestion Notification (DCQCN) on the RDMA side. These technologies embrace proactive congestion control mechanisms, where senders monitor congestion through feedback mechanisms like Explicit Congestion Notification (ECN) which employs two bits in packet headers. One bit conveys the congestion anywhere in the forward path to the receiver. The receiver then sets the other bit in packets traveling back to the sender to inform it of the congestion. ECN is a standard mechanism and is often supported or emulated in higher end data center switches.

DCTCP uses ECN to detect queue buildup in switch buffers (as opposed to actual packet drops) and modulates flow rates accordingly. Similarly, Remote Direct Memory Access over Converged Ethernet Version 2 (ROCEv2) [14] implements the RDMA protocol atop the IP layer and employs a related mechanism known as DCQCN for proactive congestion control. However, neither of them supports any QoS differentiation, an issue that we have explored extensively in [10]. This paper shows how we incorporate these capabilities in a tool supporting networked storage accesses.

2.4. Overview of DCTCP and DCQCN

In this section we provide a brief overview of DCTCP and DCQCN; a more detailed treatment is contained in our related paper [10]. Both use the notion of congestion window (CWND) which is the number of transmissions within a round-trip time (RTT). DCTCP uses the ECE bit to mark the TCP packet acknowledgments thereby indicating which packets (going in the forward direction) encountered congestion. Thus the fraction of such marked acknowledgments indicates the severity of congestion. Suppose that we have I competing TCP connections (or flows). Let $f_i(n)$ denote the fraction of marked acknowledgments for the i-th flow within its n-th window.

Since $f_i(n)$ is likely to be highly variable, it is smoothed exponentially over succeeding windows to obtain the smoothed estimate denoted as $\alpha_i(n)$.

$$\alpha_i(n) = (1-g)\alpha_i(n-1) + g f_i(n-1) \tag{1}$$

where $0 < g < 1$ is the smoothing constant with a default value of 0.5 that we determined as reasonable in most cases.

DCTCP algorithm: DCTCP reduces the window per RTT in proportion to the latest estimate of α_i such that in the limiting case of $\alpha_i = 1$, the window is halved. Let $W_i(n)$ denote the size of the nth window of flow i (i.e., the number of transmissions in this window). Then the window controlling mechanism is formulated as follows:

$$W_i(n) = W_i(n-1)\left(1 - \frac{\alpha_i}{2}\right) \tag{2}$$

DCQCN algorithm: Data Center Quantized Congestion Notification (DCQCN) is a DCTCP-like mechanism that works with the RDMA protocol (rather than TCP). It uses the PFC (priority flow control) and ETS (enhanced transmission service) features in the data center Ethernet and thus is an L2 QoS capability. PFC is limited to doing an independent flow control for each of the 8 priority classes defined by the 3 CoS (Class of Service) bits in an Ethernet frame. ETS allows for a more granular BW control across classes using a WRR (weighted round-robin) mechanism. Nevertheless, the combined PFC/ETS cannot distinguish between different destinations of flows and generates a PAUSE frame as a response to congestion for all flows in a CoS class. DCQCN uses these mechanisms with an emulated ECN to circumvent this issue. Since ROCEv2 utilizes a connectionless protocol i.e. UDP, it cannot leverage the ACK packets like in DCTCP. Hence the congestion feedback mechanism needs to be request-agnostic. The switches mark the ECN bits once congestion is detected, which the receiver NIC detects. The receiver then sends back a ROCEv2 standardized Congestion Notification Packet (CNP) to the sender NIC for every N microsecond interval. This goes on for every interval until it stops receiving ECN-marked packets. As the maximum number of CNPs received during each interval is 1, Eq.1 for DCQCN during congestion is updated as follows,

$$\alpha_i(n) = (1-g)\alpha_i(n-1) + g \tag{3}$$

The sender modifies the rate reduction factor according to Eq.3 and also stores the current rate so as to utilize it later on for flow rate recovery. It is formulated as follows,

$$RT_i(n) = RC_i(n-1) \tag{4}$$

$$RT_i(n) = RC_i(n-1)\left(1 - \alpha_i \middle/ 2\right) \tag{5}$$

The stored rate value of $RT_i(n)$ is used to increase the flow rate once the congestion episode passes. There are two phases for increasing flow rate - fast recovery and additive increase. During fast recovery, the flow rate RC_i is rapidly increased so as to reach the target value of $RT_i(n)$ in successive iterations, i.e. :

$$RC_i(n) = \left\{RT_i(n-1) + RC_i(n-1)\right\} \middle/ 2 \tag{6}$$

This fast recovery phase is followed by additive increase when the flow rate is slowly increased by an additive constant R_{AI} so as to reach the target rate as follows:

$$RT_i(n) = RT_i(n-1) + R_{AI} \tag{7}$$

$$RC_i(n) = \left\{RT_i(n-1) + RC_i(n-1)\right\} \middle/ 2 \tag{8}$$

QTCP and QRDMA: In our previous works[15, 10], we tackled QoS differentiation in the network by proposing QoS-aware versions of DCTCP and DCQCN, i.e. QTCP and QRDMA respectively. We introduced a class-specific quality factor metric, Q, to control the flow rate during congestion. Each QoS class has a specified relative throughput or latency factor, which allows us to determine Q as a ratio of actual vs. target value. Flows with

$Q > 1$ are squeezed whereas those with $Q < 1$ are allowed to expand, thereby moving all classes towards the desired allocation according to their QoS requirements. All other window modulation equations are kept the same as DCTCP (or DCQCN in the case of QRDMA) and in fact, in the absence of congestion, QTCP behaves exactly as DCTCP. The modified equations pertaining to QTCP are as follows:

$$W_i(n) = \begin{cases} W_i(n-1)+1, & \text{No ECN} \\ W_i(n-1)(1-\frac{\alpha_i}{2}), & \alpha_i \geq 0 \text{ and } Q_i \geq 1 \\ W_i(n-1)(1-\frac{\alpha_i}{2})Q_i, & \alpha_i \geq 0 \text{ and } Q_i < 1 \end{cases} \tag{9}$$

where W_i is the window size for a flow i. Similar to DCTCP, we use the value of α_i i.e., the fraction of packets that are ECN marked) along with our new metric Q_i to modulate the flow rate.

We utilize this modified version of DCTCP (i.e. QTCP) in our E2E QoS differentiation as the transport protocol. However, DCTCP can also be used if the application does not require QoS differentiation. Additionally, some applications require strict priority as mentioned in 2.3. NVMe supports strict priority as well, using a class called "Urgent", which gets nonpremptive priority over others. This is in contrast to other classes whose priority is defined through a weighted round robin (WRR) mechanism. For these applications, we provide in-network priority by reserving buffer space in the switches so as to not throttle the strict priority traffic up to a certain threshold. We henceforth include the notion of in-network priority treatment as a part of "urgent" class. In our previous work[10] we discuss this in detail and and show that this reservation is not fatal for other applications due to the Urgent application's small request sizes.

2.5. Drawbacks of contemporary simulators

In the domain of simulators, there are various tools dedicated to different aspects such as end-to-end (E2E) simulations, network simulations, and storage device simulations. On the network side, although several prominent simulators are in use such as Omnet++ [16], OPNET [17], NS3 [18], Mininet [19], etc., the most widely used and cited open-source simulator is NS3. NS3 [18] has largely been developed extensively in the networking community and emphasizes a range of networking protocols, including both wired and wireless protocols. Unfortunately, all these simulators still lack the simulation of networked storage systems and end-to-end QoS differentiation. We have made use of NS3 in creating NeSt's network stack, but have implemented additional capabilities, which in turn required a tremendous amount of effort, not to mention several bugs that needed to be tracked down and fixed.

There are several simulators in the area of storage systems, but they focus on different aspects. Some of the best known ones are simply file-system benchmarks (e.g., FIO, IOZONE, Filebench [20, 21, 22]) that create a bunch of files with some given size distribution and access them in specified ways (sequential, random, or a combination), and thus do not have much to do with underlying storage systems [23]. The same goes for the more detailed file system simulators such as Zion [24] or SimFS [25]. Others are designed for specific purposes; for example, SimSANs and SANgo concern Storage Area Networks (SANs). SimSANS is specifically intended to address the moving of legacy Fiber channel SAN to the converged Ethernet domain. Numerous storage device simulators exist, such as MQSim, StorageSim, DiskSim, VSSim, FlashSim, SimpleSSD, and some (e.g., StorageSim) can also simulate the storage hierarchy [26, 27, 28, 29, 30]. However, designing simulators to handle the entire data center storage system is a difficult task. For example, such a simulator would need to handle logical volume management (LVM) across multiple devices, data placement, storage virtualization, tiering, caching, erasure coding/replication, deduplication, encryption, power failure consistency, IO failure handling, etc. Thus, our goal is merely to design a simulator that accurately represents networked access to one or more storage devices. Other capabilities can be developed by the community on top of NeSt.

End-to-end (E2E) evaluation of datacenter applications requires a simulator that combines all the necessary components. E2E cloud simulator tools are designed to simulate the entire cloud computing environment to test applications or services. One such example is CloudSim [31]. However, a key limitation of CloudSim lies in its coarse resource modeling [32] along with its lack of a comprehensive storage endpoint. Advanced networking intricacies, such as detailed protocols, QoS (Quality of Service), and network topologies, also cannot be explicitly modeled in the standard CloudSim framework. CloudAnalyst, which is a similar tool built as an extension to CloudSim, assumes static configurations [33] but does not address the mentioned deficiencies. Hence, it does not accurately represent the E2E environment required to simulate networked storage systems.

This brings us to NeSt, which we built by taking advantage of two existing simulators, namely, NS3 for the network end and MQSim for the storage end. NeSt supports QoS differentiation at the network level, storage access protocol level, and inside the storage device itself. Additionally, it addresses the limitations of the aforementioned simulators as well. We'll delve into its unique capabilities in detail later in Section 3.

It is worth noting that in addition to simulators, there are also several emulators in existence. The key distinction is that an emulator allows unmodified code of the relevant service to be run directly within a container or a lightweight VM. Several emulators exist for the networking infrastructure such as Mininet, CORE, and SEED, and can run unmodified open-source code for various protocols and services [34, 35, 36]. However, they are not useful for networked storage applications since the code for SSD internals (e.g. FTL) is proprietary. Even for the networking part, running protocols would require executing the entire OS in the container.

3. Proposed Simulator

An E2E storage access scenario (right from the host to the storage device and back to the host) follows the steps given in Fig.1. There are 3 major components in NeSt's E2E path:

- Network Module: This consists of the host end and the network combined into a single module. It simulates both the sender and receive side network path with a detailed implementation of the network stack, including L2/L3 switches.
- Storage Module: This simulates SSDs along with the storage access protocol and is discussed in detail in 3.2. Additionally, it also supports the simulation of PM accesses, which is discussed in 3.3.
- Interfacing Module: This module acts as the connector between the network module and the storage module and is explained in detail in 3.5

Our proposed simulator, NeSt, utilizes the concept of nodes to simulate endpoints such as hosts and storage servers. Each host has the functionality to house different types of applications. The links connecting nodes are bidirectional and the transport protocol used can be configured. Supported transport options are datacenter versions of TCP, RDMA, and QoS-aware versions of TCP and RDMA (i.e., QTCP and QRDMA). Other protocols

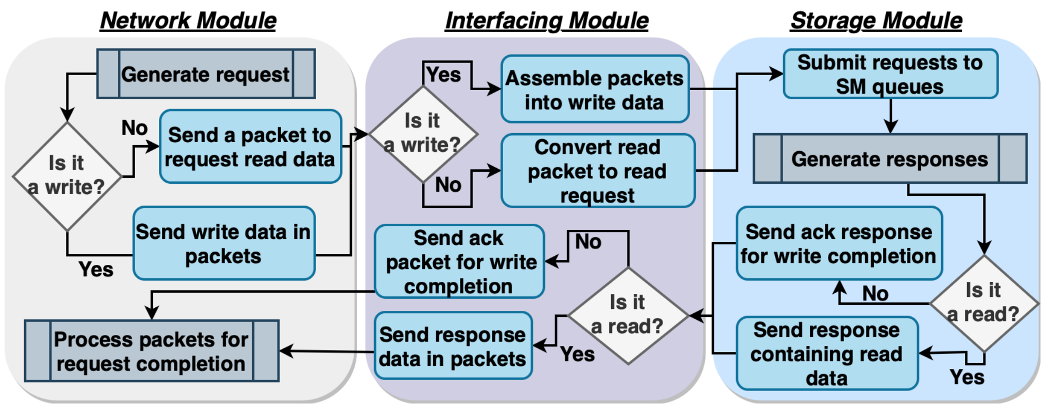

Network Module
Interfacing Module
Storage Module
Generate request
Is it a write?
No
Send a packet to request read data
Send write data in packets
Yes
Process packets for request completion
Yes
Assemble packets into write data
Is it a write?
No
Convert read packet to read request
Send ack packet for write completion
No
Is it a read?
Send response data in packets
Yes
Submit requests to SM queues
Generate responses
Send ack response for write completion
No
Is it a read?
Send response containing read data
Yes

Figure 1: E2E Flowchart for NeSt

built into NS3 should also be usable but have not been included in our evaluation in 5.

As shown in Fig.1, the host first generates the request and the request is broken into equal sized packets in case of a write (i.e. the write data) or a single packet is sent in case of a read (i.e. the read request). The Interfacing Module assembles write data packets and submits the write data (or the read request) to the Storage Module, which in turn generates and sends back the responses to the Interfacing Module. This, in turn, breaks the responses into equal sized packets in case of a read while it sends a write completion acknowledgment back to the host. The host processes all response packets to mark request completion.

3.1. Architecture of MQSim

NeSt's storage module architecture is based on the previously mentioned popular SSD simulator, MQSim. As shown in Figure 2, it comprises the Host Interface which fetches requests from the interfacing protocol (i.e. NVMe) and translates them to device commands (using the Request Fetch Unit and the Input Stream manager respectively). It then checks if the data is present in the DRAM cache present in the SSD using the Data Cache Manager. The Flash Translation Layer (FTL) then manages the mapping between logical addresses used by the host and physical flash memory locations using its Address Mapping Unit. The Background Operations Unit manages activities such as garbage collection and wear leveling by communicating with the FTL and the SSD backend. Finally, the Transaction Scheduling Unit fetches and

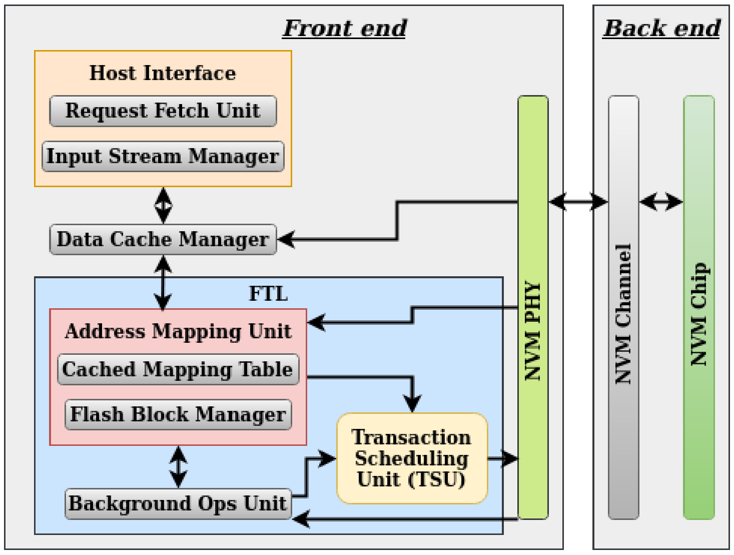


Figure 2: Architecture of MQSim

serves requests from the FTL by accessing the SSD backend using the NVM channel.

Additionally, MQSim's evaluation has revealed that it closely approximates the performance of actual SSDs, with response time errors averaging between 11% and 18% [37]. It allows for the evaluation and assessment of different types of SSD performance by simulating various configurations that can be managed by user-settable parameters. However, it is crucial to acknowledge the limitations inherent to MQSim, including incomplete support for NVMe's Weighted Round Robin queue arbitration mechanism, its inability to simulate multiple devices, and the absence of support for behavioral simulation of Persistent Memory technologies and remote device access.

3.2. Multiple Devices with Multiple Configurations

Our storage end simulation of SSDs is based off of the widely used SSD simulator - MQSim. Since MQSim only supports a single SSD, we extended it by vectorizing all data structures, objects, and function calls while making changes wherever necessary to accommodate object-oriented programming concepts. Furthermore, we also added a new feature that ensures each device in the array of devices can support different device configurations. For example, we can simulate a SLC SSD, MLC SSD, and a TLC SSD at the same time, with each of them having differing characteristics. NeSt also provides the additional functionality of reading multiple configuration files during runtime. Further, the number of devices is a settable parameter. We

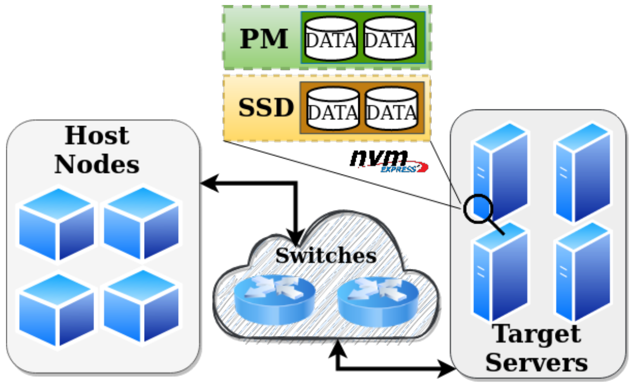


Figure 3: High Level Architecture of NeSt

have evaluated NeSt with a maximum of 30 devices (shown later in 5.4.4), however, the number of devices can be increased depending on the specifications of the system on which NeSt is run.

3.3. Simulation of Persistent Memory Accesses

As mentioned before, PM devices are at least 3-5 times faster than traditional NVMe SSDs and may have a very different internal architecture depending on the technology [38]. In particular, Intel Optane has a DRAM-like structure involving multiple ranks and banks. We provide a behavioral simulation of PM accesses by switching off access to the SSD backend and extending the DRAM buffer module present inside our storage module (i.e. the buffer used inside an SSD to cache frequently accessed data). We do not simulate the internal architecture of the PM device and do not believe that is necessary in NeST. Additionally, we have preserved the DRAM caching capabilities of an SSD device simulation by introducing a flag that helps in determining if this DRAM module is being used as an SSD cache or to simulate PM accesses. A request that is tagged as Urgent is directed towards our PM module. The request (whether it be a read request or write data) is resolved in our modified DRAM buffer cache and the response (i.e. read data or write acknowledgment) is sent out, thus resulting in an access performance that is faster than SSDs and comparable to PM devices. Our simulation of the PM accesses allows it to serve both storage requests (i.e. in request sizes ranging from 512 bytes to 4KB blocks) and memory requests (i.e. in request sizes ranging from 2-4 cacheline/64-128 bytes). Remote applications pertaining to the former mode of access are either throughput or latency sensitive while the latter mode applies to ultra-low latency-sensitive applications specifically. In our evaluation of this feature, we consider large transfer requests from both throughput and latency-sensitive applications. The overall end-to-end architecture of NeSt (with NVMe devices at the target end) is shown in Fig.3 where we have host nodes (housing the applications) accessing target servers via network switches. These target servers contain our simulation of the PM accesses and the simulation of the SSD device, connected by the NVMe storage access protocol. Since the NeSt model currently has no representation of CPU, the only real distinction between storage and memory accesses is their performance, depending on the transfer sizes and the latency parameters. As we have already explored small transfer memory accesses previously[10], in this work we evaluate the performance of large transfer storage accesses from PM applications.

3.4. QoS Differentiation in the Storage Module

QoS differentiation in the NVMe level uses Weighted Round Robin (WRR) queue arbitration on the submission queues (i.e. the interfacing queues). Even though existing simulators claim to support this feature, their implementations are traditionally a naive workaround of the actual mechanism. We made the necessary changes to accommodate this queue arbitration feature. We also extended the WRR implementation to the in-device queues to ensure that there is QoS differentiation inside the device as well, thus making sure that the QoS differentiation is truly end-to-end, except that it does not deal with managing the background activities inside the SSD. Our previous work[39, 40] has looked into tackling these background latencies and could be incorporated in NeSt but is not done currently.

3.5. Interfacing between Network and Storage

To simulate the datacenter environment, we need to interface the host requests arriving via the network, with the storage end. We introduced a separate *Interfacing Module* which lies between the Network and Storage Modules. This module contains a pair of queues, i.e. the request and response queues. The former carries the write data and read requests towards the storage device while the latter carries the write acknowledgments and the read data back to the hosts. The batch size for servicing the request and response queues is kept as 1 to ensure that there are no queuing delays caused by the Interfacing Module. This is because this module is a simulator component and not an actual E2E path component. Hence it is desired that its effect on the E2E performance of a request be minimal if any.

3.6. Trace Replay Module

A limitation present in network simulators (ex. NS3) is the absence of replaying storage block traces. This is because network simulators aim to simulate and capture packet traces. Due to the absence of a target device endpoint, these simulators are not concerned with block requests. NeSt addresses this by introducing a sub-module in the network module that reads storage trace files and breaks individual requests into packets. It reads a request and inserts its corresponding information into the event tree at the specified timestamp (which is read from the trace file too). The request is triggered at the given timestamp by the event tree and the next request information is again inserted into the event tree. This continues until the entire trace file has been read. Additionally, this module also contains the capability to scale entire simulations. For example, a one hour long trace can be decreased to a 6 minute long trace or increased to a ten hour long trace. This scaling factor can be modified according to the user requirements. This feature helps in introducing complex congestion scenarios in realistic datacenter environments or to reduce the length of long running simulations. We have utilized this feature in our evaluations later on.

3.7. Time Synchronization between Network and Storage

Event-driven simulations require all events to be properly ordered in simulation time, else the interactions between the events could lead to unpredictable results. This is not a problem with a single simulator designed from scratch; however, it can become a problem when integrating two distinct simulators. NeSt attempts to integrate the simulation engines of NS3 and MQSim and thus time synchronization between the two becomes important.

We solved this synchronization issue by using a single clock and event tree, i.e. the network end tree and clock. We leveraged the network end event tree and clock by making modifications in the storage simulation module. These changes ensured that the storage module followed the network module's event clock and all its events were inserted in the single event tree, thus preserving the relationship between dependant events.

However, this is not enough for the seamless handling of requests between both the network and storage end. We also introduced a new host module on the storage end. This module remains dormant unless triggered by the interfacing module. The triggering event occurs whenever the network end finishes processing a request, i.e. when the request reaches the endpoint node via the switches and pushes the processed request into the interfacing request queue. The newly introduced host module then generates this request from the interface queue by converting it into the format recognized by the device, following which it pushes it into the submission queue of the NVMe device.

4. Comparison with existing simulators

	Network Simulators	***Storage Simulators***	***E2E Simulators***	***NeSt***
Datacenter Networks	☑	☐	☑	☑
Transport Protocol Options	☑	☐	☑	☑
Comprehensive Storage Architecture	☐	☑	☐	☑
PM Access Simulation	☐	☐	☐	☑
Multi-device with multi-configurations	☐	☐	☐	☑
Storage Protocol Level QoS Diff	☐	☑	☐	☑
Network Level QoS Diff	☑	☑	☑	☑
Device Level QoS Diff	☐	☐	☐	☑

Figure 4: Comparison of Different Open Source Simulators

As mentioned in 2.5, open-source simulators are widely utilized by the research community to capture components crucial to a datacenter environment. These simulators can be broadly classified into three different types - network simulators (ex. NS3 [18], Omnet++ [16]), storage simulators (ex. MQSim [26], SimpleSSD [30]), and E2E simulators (ex. Cloudsim [31]). In Fig.4 we compare the features required for an end-to-end QoS differentiating datacenter storage simulator with the existing simulators that fall under the mentioned types of simulators. The network simulators such as NS3 and Omnet++ can simulate the network but not the target device endpoint. Some of the storage related limitations of network simulators are addressed in storage simulators such as MQSim and SimpleSSD, which are SSD simulators.

However, storage simulators do not consider accesses in a datacenter context and also do not support the simulation of multiple devices with varying configurations. Additionally, they do not consider the PM accesses. Some E2E simulators such as CloudSim have been enhanced by research teams to support SSD accesses and thus in a way simulate end-to-end access of the target device. However, it does not simulate the storage access protocol (i.e., NVMe) while QoS differentiation is also absent. NeSt addresses all these limitations of the aforementioned simulators by supporting all the mentioned necessary features as shown in Fig.4.

5. Evaluation

5.1. Comparison with Real Systems

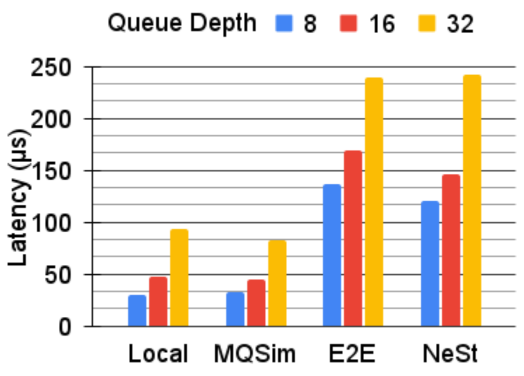


Figure 5: Comparison between simulation and real environment

The first step in evaluating a simulator is to compare its performance with a real environment. In this section, we compare the latency observed in the following four scenarios with respect to SSD storage access:

- Local access to an NVMe SSD using a mixed read-write workload generated using iperf
- Remote access to the same SSD (with NVMe-oF) using the same workload with TCP being used as the transport protocol protocol
- Simulated access to an SSD using MQSim
- E2E simulated access using NeSt

The SSD used for the real experiments is a Samsung 970 EVO Plus while the SSD used for the simulation experiments mirrors the same parameters exhibited with the Samsung SSD. MQSim provides a comprehensive settable parameter list pertaining to the SSD being simulated. We utilize this feature to simulate the same SSD as the real experiments with increasing queue depth. In the real experiments, SPDK [41] was used for both local and E2E access to the device. For the remote access, we connected two Dell Precision Tower workstations using a switch, with the host workstation requesting data present in the SSD on the target workstation using SPDK. All links are using 100Gbps.

We first compare the local access to the SSD using MQSim. We notice that both local and MQSim performance are nearly identical for all the queue depths considered, with the latency observed ranging from 40-90μs. This is due to MQSim's accurate and comprehensive replication of the SSD frontend and backend, which includes details of the NAND architecture present in an SSD along with queuing at different levels. This further makes it an ideal candidate for use in NeSt as the simulated target device. We now compare NeSt's performance with real E2E storage access. In this case, NeSt's architecture resembles a host application requesting data from a storage server with a single SSD (with the same parameters) via an intermediate switch. All links are 100Gbps and the transport protocol used for NVMe-oF is also TCP. We notice that NeSt's performance is also similar to the behavior exhibited by the real experiment with the latency ranging from 120-190μs. The slight difference in latency is due to the lack of kernel involvement, thus this latency is not modeled by the NeSt. Thus we can say that NeSt accurately models a real end-to-end scenario. However, since these scenarios are without congestion, the carried throughput is not interesting. We discuss throughput evaluations for congested scenarios in subsequent sections.

5.2. E2E Throughput Evaluation

In this section, we look into NeSt's capability in guaranteeing E2E throughput differentiation. We consider network throughput and storage end (i.e. NVMe + device end) throughput as two separate components and show how both of them provide throughput differentiation in order to guarantee E2E requirements. The three workloads (for High, Medium, and Low QoS classes) have been generated using NeSt's built-in traffic generator. We evaluated a read-only workload, a write-only workload, and a read and write

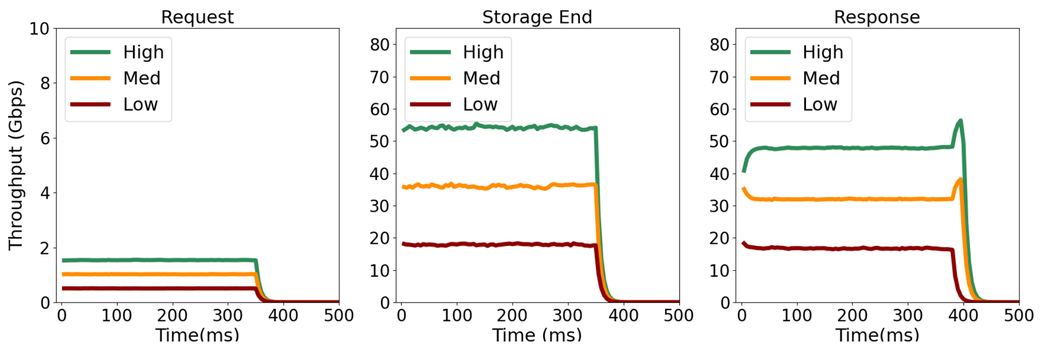


Figure 6: Evaluation of a throughput sensitive read only workload

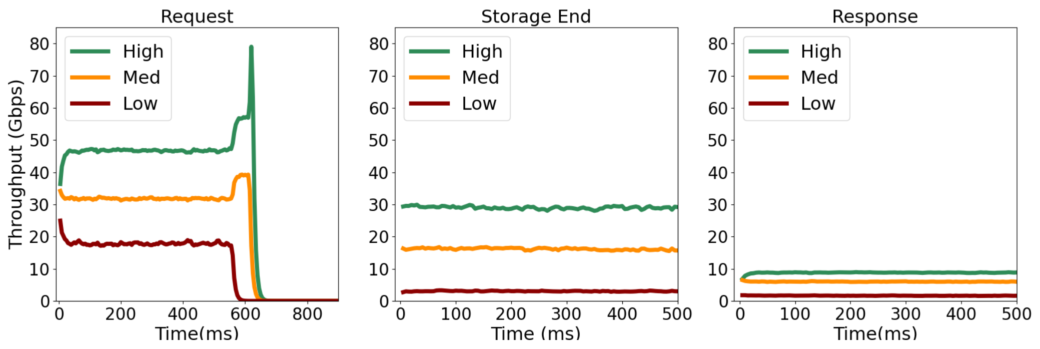


Figure 7: Evaluation of a throughput sensitive write only workload

mixed workload. The workloads follow an exponential request interarrival time distribution and the endpoint consists of ten different storage servers.

For the read-only workload in Fig.6, the bottleneck is on the backward receive link due to the read data being transferred from the storage end exceeding the bottleneck link capacity of 100Gbps. The High, Medium, and Low QoS classes read data at a rate of 90Gbps, 60Gbps, and 30Gbps respectively. This causes congestion and QTCP is triggered, which we observe at the response end in Fig.6. The storage end also provides QoS differentiation due to the NVMe WRR protocol coupled with the proposed modification in 3.4. The three classes receive throughput differentiation in the ratio of 3:2:1. This shows that the E2E throughput differentiation is provided according to the application's QoS class. For the write-only workload (in Fig.8) the bottleneck is on the sender side due to the applications' write data congesting the sender bottleneck link of 100Gbps. In this case too we notice that the QoS requirements are respected similar to the read-only workload. The

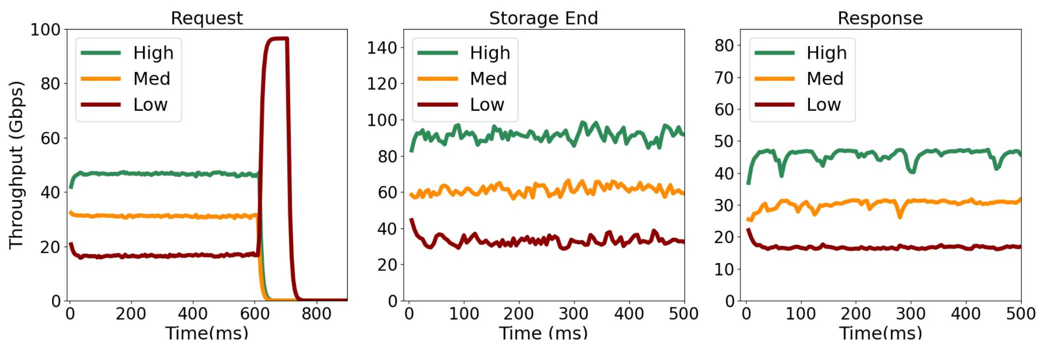
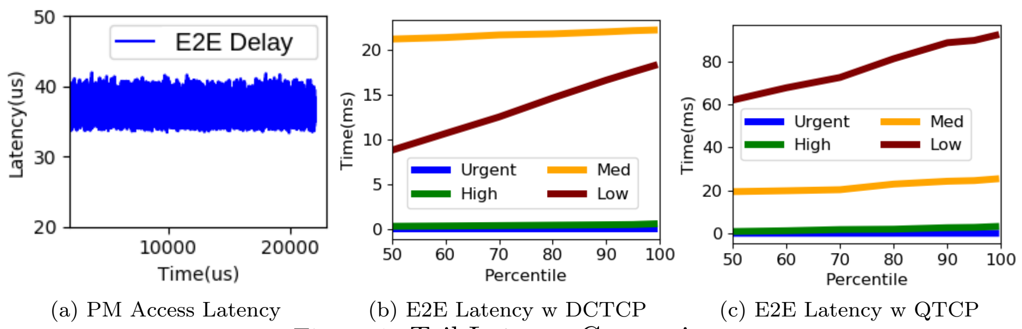


Figure 8: Evaluation of a throughput sensitive read write workload

storage end provides the differentiation too.

In the case of the read write workload, we utilized the combination of the workloads used in Fig.6 and Fig.8. This creates a pathological scenario with both sender-side and response-side links congested. However, as seen in Fig.8, NeSt shows E2E throughput differentiation even in this case. With all 3 components guaranteeing throughput requirements, i.e. request path, storage end, and response path. The throughput is once again divided in the ratio of 3:2:1. This shows that NeSt's throughput differentiation capabilities (i.e. QTCP in the network end, NVMe WRR, and SSD Device-level Differentiation) adequately provide E2E throughput differentiation irrespective of the workload type. We evaluate applications with PM access in subsequent sections.

5.3. E2E Latency Evaluation



(a) PM Access Latency (b) E2E Latency w DCTCP (c) E2E Latency w QTCP
Figure 9: Tail Latency Comparison

We now look at E2E latency evaluation for NeSt. First, we test NeSt's simulation of PM accesses. Fig.9a shows the PM access latencies exhibited by urgent class requests. We chose the arrival process of the requests to PM as 4KB in size because small sizes would be infeasible due to large network latency. We observe that the latency ranges from 35-40μs which is surprisingly low considering that this is the E2E latency of a storage request. However, our in-network prioritization using reserved buffer space in the switches helps in achieving "Urgent" class status for these applications.

We now consider a mixed traffic scenario comprising PM applications and other storage applications under congestion. Clearly, it is not possible to meet the latency bounds for both types of traffic since the latency will continue to go up under congestion. Hence we consider a transient congestion episode that lasts for 2 seconds in Fig.9b and Fig.9c. The former uses DCTCP as the transport protocol while the latter uses QTCP. The workloads for the High, Med, and Low classes follow an exponential request interarrival time distribution with High pushing 90Gbps, Medium 60Gbps, and Low 30Gbps worth of traffic.

We observe in Fig.9b that DCTCP fails to provide correct QoS differentiation during the transient congestion episode, with the Medium QoS class performing far worse than the Low QoS class. However, this is remedied by using QTCP in Fig.9c with all four QoS classes receiving relative QoS differentiation. It is interesting to note the exact ratios of latency are not maintained as it is difficult to control background latencies in the SSD device (caused due to the SSD's internal architecture as mentioned in 3.1) along with queuing in the network. However, we can still provide relative differentiation as we have shown in Fig.9c. Finally, we can also see that both Urgent and High classes get their differentiation in both cases. This is due to the fact that both these classes have fewer number of requests with smaller request sizes. Our evaluation in this section also shows NeSt's capability in evaluating remote applications running on different transport protocols, ex. DCTCP.

5.4. Evaluating Realistic Scenarios

In this section, we utilize NeSt's trace replay module(explained in 3.6) to evaluate other offered features. Before we dive into the results of our evaluation, we first discuss the workloads we have used.

5.4.1. Workload Characteristics

For testing the storage trace replaying functionality of NeSt, we utilized the traces published by Tencent at ATC, 2020[42]. These traces are a col-

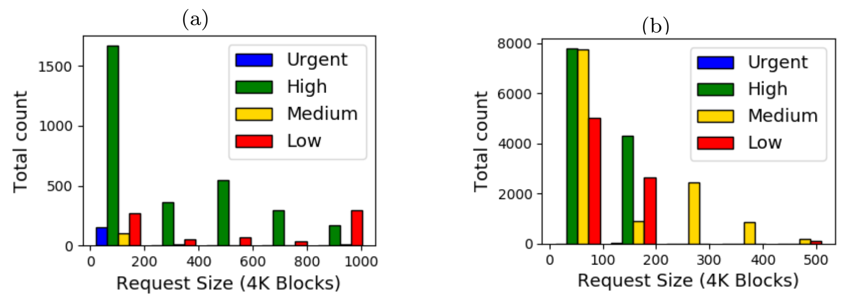


Figure 10: Workload (a) read and (b) write distribution

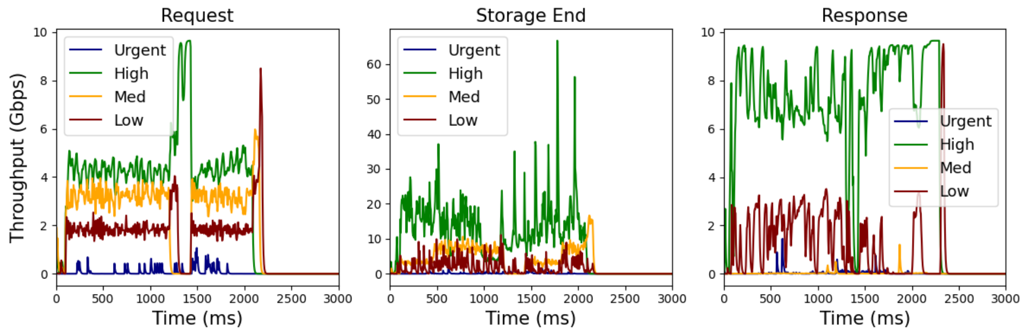


Figure 11: Throughput differentiation of a real workload

lection of six day long I/O traces from a production cloud block storage system. The cloud block storage system contains tens of thousands of cloud disks. These traces push total traffic worth 3-4Gbps and hence we utilized the scaling feature present in the trace replay module to address the slow interarrival time of the traces. This helps in creating congestion scenarios in 10Gbps links.

We selected traces for four different servers depending on their read and write distribution as shown in Fig.10a and Fig.10b respectively. The Urgent class workload has the lowest read and write size counts and, thus the lowest storage and network load. This is characteristic of applications that require the highest priority treatment[43]. We also chose workloads depending on their read and write mix, for example, the medium class workload is write intensive while high and low class contain a good mix of both read and write requests.

5.4.2. PM Access Evaluation

We test how NeSt's PM module performs when used as a storage device, i.e. when accessed by throughput-sensitive applications. We utilize the work-

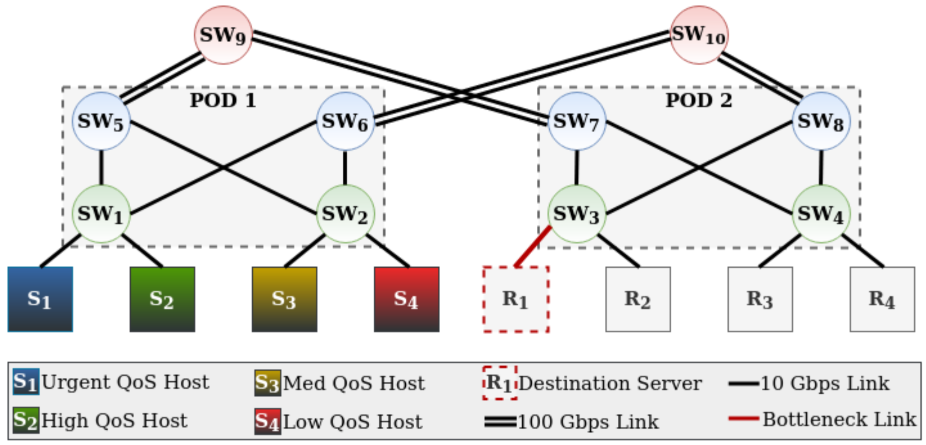


Figure 12: Fat Tree Topology

loads discussed in 5.4.1 and preserve the same experimental setup used in 5.2. The bottleneck link is changed to 10Gbps to cause a congestion episode. We observe in Fig.11 that the Urgent traffic (i.e. the application accessing the PM module) receives its desired throughput. We utilize the buffer reservation for PM traffic as explained in 2.4 to give it the highest priority in the network. Similarly, the storage device itself also treats it as Urgent priority. We notice that slight peaks in traffic for the Urgent traffic result in traffic pertaining to other QoS classes making way for it. This reactive behavior to traffic changes is maintained even when a certain QoS class exits the simulation and re-enters at a later time. For example, in Fig.11 we see that the Medium class traffic (which is a write heavy workload) stops performing writes at around the 1250ms mark and restarts at around the 1500ms mark. As soon as it exits the simulation, both the High and Low classes grab onto the available bandwidth according to their respective QoS classes. Again, we see that the Low class also exits at around the 1300ms mark and hence the High class traffic grabs onto the available bandwidth again. As soon as both the Medium and Low classes re-enter the datacenter, the High class traffic is squeezed while also respecting its QoS class. On the storage end, the throughput differentiation is not as clear due to no bottleneck present in the storage end, thus all classes receive their required throughput. On the response end, only three classes vie for the available bandwidth due to the Medium class being write heavy.

5.4.3. Fat Tree Evaluation

We now look at how NeSt performs in a real datacenter topology. We use the fat tree topology as it is still widely used in datacenters. Our fat tree(as

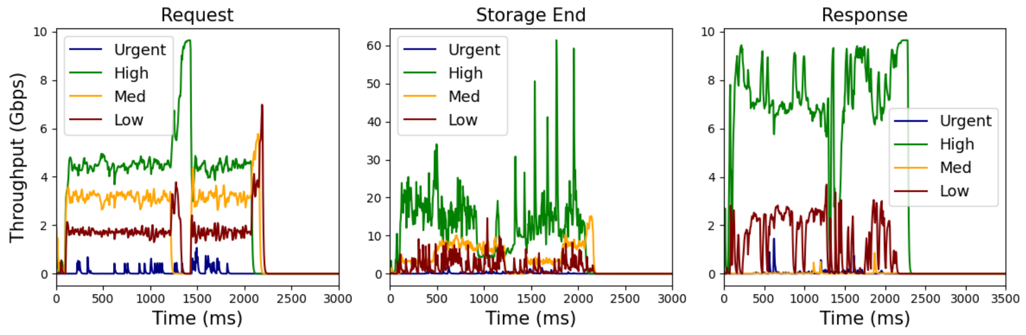
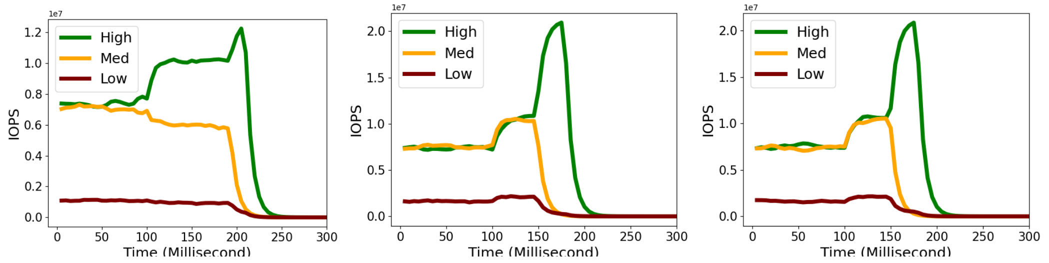


Figure 13: Throughput differentiation in a fat tree topology

shown in Fig.12) has 2 pods with Pod 1 being connected to the four host systems. The host systems request data from the storage servers R1, R2, R3, and R4 in Pod 2. The core level links are 100Gbps while the aggregation and edge level links are 10Gbps. We use the same workloads as in Fig.11. In this case, we observe that the behavior is nearly identical to Fig.11. Both Fig.13 and Fig.11 show very similar behavior and thus the QoS abiding techniques used by NeSt achieve the required differentiation for realistic data center network topologies as well.

5.4.4. Multi Device Simulation



(a) No. of SSDs = 3 (b) No. of SSDs = 10 (c) No. of SSDs = 20

Figure 14: Storage Bottleneck Comparison

We now look into NeSt's feature of simulating multiple NVMe SSD devices. A bottleneck in the storage server is generally much less likely than the network. Thus the storage end will usually respect the throughput provided by the network. In the case when the network is QoS agnostic, the storage device also turns QoS agnostic. However, SSD bandwidth could become a bottleneck if only a few large-capacity devices are used to support many applications. We evaluate three different scenarios in Fig.14 with different number of SSDs at the storage end and utilize the workload used in Fig.7 to create a bottlenecked forward link scenario. The network is QoS agnostic in this experiment, hence the incoming traffic does not respect the QoS classes and it is up to the storage to provide QoS differentiation. In Fig.14a, the number of devices is 3 and we can see that the storage throughput differentiation (as mentioned in 3.4) gradually kicks in as the bottleneck scenario arises. On increasing the number of devices to 10 and 20 in Fig.14b and Fig.14c respectively, the bottleneck disappears. In both these cases, the storage end still tries to provide differentiation by squeezing the Low class traffic, however, it fails to provide differentiation between the High and Medium classes.

6. Conclusions and Future Work

In this paper, we presented the design and evaluation of a simulator for networked storage systems in enterprise systems. The simulator, named NeSt, is specifically designed to simulate remote access to a group of flash-based storage devices (i.e., SSDs) accessed via the increasingly dominant NVMe protocol, carried over a TCP or RDMA-based network transport. The key innovation in NeSt is end-to-end (E2E) QoS differentiation across multiple classes of service. This is done by conveying the class designation end to end and using it to consistently but independently apply the differentiation in each segment of the path. NeSt contains the ability to provide end-to-end QoS differentiation under a variety of traffic situations including both large block accesses (4KB or larger) to traditional storage and small accesses (a few cachelines) to the emerging persistent memory devices.

We will open-source NeSt to enable the storage community at large to not only use it but also enhance it in multiple possible directions. One very useful enhancement is to implement the logical volume manager (LVM) functionality whereby the storage can be allocated flexibly across multiple SSDs. A related aspect is coupling LVM with high-performance data layouts such as striping blocks across multiple SSDs. With the emergence of computational storage, it would be interesting to explore some generic functionalities in optimizing the storage transfers based on the end-to-end class hinting mechanism discussed in this paper. Intelligent prefetching is one such function where the aggressiveness of the prefetching could be QoS class-dependent. This prefetching may be limited only to the SSD cache or PM, from where it could be fetched directly on demand for all but the data with a high reuse rate. This would be an extension of our FussyCache concept [44]. Many other extensions such as inclusion of a CPU model, transactional PM, erasure coding, and support for QRDMA, are also possible and would be highly useful for emerging storage applications.

References

[1] I. Gorton, P. Greenfield, A. Szalay, R. Williams, Data-intensive computing in the 21st century, Computer 41 (4) (2008) 30–32.

[2] R. Davis, The network is the new storage bottleneck, https://www.datanami.com/2016/11/10/network-new-storage-bottleneck/ (2016).

[3] B. Tallis, The samsung 970 evo plus (250gb, 1tb) nvme ssd review, https://www.anandtech.com/show/13761/the-samsung-970-evo-plus-ssd-review/5 (2019).

[4] S. Zheng, M. Hoseinzadeh, S. Swanson, Ziggurat: A tiered file system for non-volatile main memories and disks, in: Proceedings of the 17th USENIX Conference on File and Storage Technologies, FAST'19, USENIX Association, USA, 2019, p. 207–219.

[5] Nvm express base specification, rev 1.4, https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf (June 2019).

[6] J. D. Allen, J. Metz, The evolution and future of nvme, Webminar, Jan 2018 (2018).

[7] Nvm express over fabrics revision 1.1, https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1-2019.10.22-Ratified.pdf (2019).

[8] Introduction to nvme technology, https://www.osr.com/nt-insider/2014-issue4/introduction-nvme-technology/ (2014).

[9] S. Akram, Performance evaluation of intel optane memory for managed workloads, ACM Transactions on Architecture and Code Optimization (TACO) 18 (3) (2021) 1–26.

[10] J. Gupta, K. Kant, A. Pal, J. Biswas, Configuring and coordinating end-to-end qos for emerging storage infrastructure, ACM Trans. Model. Perform. Eval. Comput. Syst. 9 (1) (jan 2024). doi:10.1145/3631606. URL `https://doi.org/10.1145/3631606`

[11] B. Wang, Z. Qi, R. Ma, H. Guan, A. V. Vasilakos, A survey on data center networking for cloud computing, Computer Networks 91 (2015) 528–547.

[12] What is Persistent Memory Region? - NVM Express — nvmexpress.org, `https://nvmexpress.org/faq-items/what-is-persistent-memory-region/`, [Accessed 28-01-2024].

[13] J. Shin, J. W. Kim, C.-C. Kuo, Quality-of-service mapping mechanism for packet video in differentiated services network, IEEE Transactions on Multimedia 3 (2) (2001) 219–231.

[14] N. Schelten, F. Steinert, A. Schulte, B. Stabernack, A high-throughput, resource-efficient implementation of the rocev2 remote dma protocol for network-attached hardware accelerators, in: 2020 International Conference on Field-Programmable Technology (ICFPT), IEEE, 2020, pp. 241–249.

[15] A. P. Joyanta Biswas, Krishna Kant, Qos aware tcp for data center network (qtcp), Proc. of LCN (Oct 2021). doi:10.1109/LCN52139.2021.9524967.

[16] A. Varga, R. Hornig, An overview of the omnet++ simulation environment, in: 1st International ICST Conference on Simulation Tools and Techniques for Communications, Networks and Systems, 2010.

[17] J. Prokkola, Opnet-network simulator, URL http://www. telecomlab. oulu. fi/kurssit/521365A tietoliikennetekniikan simuloinnit ja tyokalut/Opnet esittely 7 (2006).

[18] G. F. Riley, T. R. Henderson, The ns-3 network simulator., in: Modeling and Tools for Network Simulation, Springer, 2010, pp. 15–34. URL `http://dblp.uni-trier.de/db/books/collections/Wehrle2010.html#RileyH10`

[19] F. Keti, S. Askar, Emulation of software defined networks using mininet in different simulation environments, in: 2015 6th International Conference on Intelligent Systems, Modelling and Simulation, IEEE, 2015, pp. 205–210.

[20] Welcome to FIO&x2019;s documentation! &x2014; fio 3.36 documentation — fio.readthedocs.io, `https://fio.readthedocs.io/en/latest/index.html`, [Accessed 03-02-2024].

[21] W. D. Norcott, Iozone filesystem benchmark, http://www. iozone. org/ (2003).

[22] R. McDougall, J. Mauro, Filebench (2005).

[23] S. Li, D. Li, D. Wu, X. Chen, Nvmfs-iozone: Performance evaluation for the new nvmm-based file systems, in: Proceedings of the 13th ACM International Systems and Storage Conference, 2020, pp. 87–97.

[24] F. Paladin, D. R. Adams, et al., Zion file system simulator, Journal of Computer and Communications 4 (04) (2016) 10.

[25] S. Di Girolamo, P. Schmid, T. C. Schulthess, T. Hoefler, Simfs: a simulation data virtualizing file system interface, in: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2019, pp. 621–630.

[26] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, O. Mutlu, Mqsim: A framework for enabling realistic studies of modern multi-queue ssd devices, in: Proceedings of the 16th USENIX Conference on File and Storage Technologies, FAST'18, USENIX Association, USA, 2018, p. 49–65.

[27] J. Yoo, Y. Won, J. Hwang, S. Kang, J. Choi, S. Yoon, J. Cha, Vssim: Virtual machine based ssd simulator, in: 2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST), IEEE, 2013, pp. 1–14.

[28] Y. Kim, B. Tauras, A. Gupta, B. Urgaonkar, Flashsim: A simulator for nand flash-based solid-state drives, in: 2009 First International Conference on Advances in System Simulation, IEEE, 2009, pp. 125–131.

[29] J. S. Bucy, G. R. Ganger, et al., The DiskSim simulation environment version 3.0 reference manual, School of Computer Science, Carnegie Mellon University, 2003.

[30] M. Jung, J. Zhang, A. Abulila, M. Kwon, N. Shahidi, J. Shalf, N. S. Kim, M. Kandemir, Simplessd: Modeling solid state drives for holistic system simulation, IEEE Computer Architecture Letters 17 (1) (2017) 37–41.

[31] T. Goyal, A. Singh, A. Agrawal, Cloudsim: simulator for cloud computing infrastructure and modeling, Procedia Engineering 38 (2012) 3566–3572.

[32] M. R. Chowdhury, M. R. Mahmud, R. M. Rahman, Implementation and performance analysis of various vm placement strategies in cloudsim, Journal of Cloud Computing 4 (1) (2015) 1–21.

[33] B. Wickremasinghe, R. N. Calheiros, R. Buyya, Cloudanalyst: A cloudsim-based visual modeller for analysing cloud computing environments and applications, in: 2010 24th IEEE international conference on advanced information networking and applications, IEEE, 2010, pp. 446–452.

[34] P. B. Patil, K. S. Bhagat, D. Kirange, S. Patil, Software defined networks using mininet, Int. J. Recent Technol. and Eng 9 (1) (2020) 843–849.

[35] J. Ahrenholz, C. Danilov, T. R. Henderson, J. H. Kim, Core: A real-time network emulator, in: MILCOM 2008-2008 IEEE Military Communications Conference, IEEE, 2008, pp. 1–7.

[36] W. Du, H. Zeng, K. Won, Seed emulator: an internet emulator for research and education, in: Proceedings of the 21st ACM Workshop on Hot Topics in Networks, 2022, pp. 101–107.

[37] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, O. Mutlu, {MQSim}: A framework for enabling realistic studies of modern {Multi-Queue}{SSD} devices, in: 16th USENIX Conference on File and Storage Technologies (FAST 18), 2018, pp. 49–66.

[38] M. Böther, O. Kißig, L. Benson, T. Rabl, Drop it in like it's hot: An analysis of persistent memory as a drop-in replacement for nvme ssds, in:

Proceedings of the 17th International Workshop on Data Management on New Hardware, 2021, pp. 1–8.

[39] T. Roy, J. Gupta, K. Kant, A. Pal, D. Minturn, A. Tavvakol, Managing ssd tail latency with plm, Proc. of NAS Conference (Oct 2021). doi:10.1109/NAS51552.2021.9605470.

[40] T. Roy, J. Gupta, K. Kant, A. Pal, D. Minturn, Plmlight: Emulating predictable latency mode in regular ssds, Proc. of IEEE NCA conference (Nov 2021). doi:10.1109/NCA53618.2021.9685772.

[41] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, L. E. Paul, Spdk: A development kit to build high performance storage applications, in: 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2017, pp. 154–161.

[42] Y. Zhang, P. Huang, K. Zhou, H. Wang, J. Hu, Y. Ji, B. Cheng, OSCA: An Online-Model based cache allocation scheme in cloud block storage systems, in: 2020 USENIX Annual Technical Conference (USENIX ATC 20), USENIX Association, 2020, pp. 785–798.
URL `https://www.usenix.org/conference/atc20/presentation/zhang-yu`

[43] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, G. R. Ganger, Prioritymeister: Tail latency qos for shared networked storage, in: Proceedings of the ACM Symposium on Cloud Computing, SOCC '14, Association for Computing Machinery, New York, NY, USA, 2014, p. 1–14. doi:10.1145/2670979.2671008.
URL `https://doi.org/10.1145/2670979.2671008`

[44] J. Gupta, K. Kant, Fussycache: A caching mechanism for emerging storage hierarchies, Proc. of IEEE CloudCom (Dec 2020). doi:10.1109/CloudCom49646.2020.00010.