Configuring and Coordinating End-to-end QoS for Emerging Storage Infrastructure

JIT GUPTA and KRISHNA KANT, Computer and Information Sciences, Temple University, USA
AMITANGSHU PAL, Computer Science and Engineering, Indian Institute of Technology Kanpur, India
JOYANTA BISWAS, Computer and Information Sciences, Temple University, USA

Modern data center storage systems are invariably networked to allow for consolidation and flexible management of storage. They also include high-performance storage devices based on flash or other emerging technologies, generally accessed through low-latency and high-throughput protocols such as Non-volatile Memory Express (NVMe) (or its derivatives) carried over the network. With the increasing complexity and data-centric nature of the applications, properly configuring the quality of service (QoS) for the storage path has become crucial for ensuring the desired application performance. Such QoS is substantially influenced by the QoS in the network path, in the access protocol, and in the storage device. In this article, we define a new transport-level QoS mechanism for the network segment and demonstrate how it can augment and coordinate with the access-level QoS mechanism defined for NVMe, and a similar QoS mechanism configured in the device. We show that the transport QoS mechanism not only provides the desired QoS to different classes of storage accesses but is also able to protect the access to the shared persistent memory devices located along with the storage but requiring much lower latency than storage. We demonstrate that a proper coordinated configuration of the three QoS's on the path is crucial to achieve the desired differentiation, depending on where the bottlenecks appear.

CCS Concepts: • **Networks** → **Network performance evaluation**; **Network simulations**; • **Computer systems organization** → **Distributed architectures;**

Additional Key Words and Phrases: NVMe, QoS, persistent memory, data center TCP, RDMA, latency

ACM Reference format:
Jit Gupta, Krishna Kant, Amitangshu Pal, and Joyanta Biswas. 2024. Configuring and Coordinating End-to-end QoS for Emerging Storage Infrastructure. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 9, 1, Article 4 (January 2024), 32 pages.
https://doi.org/10.1145/3631606

1 INTRODUCTION

The insatiable demand for accessing ever more data with very low latency makes the storage systems central to the performance of nearly all enterprise applications. Fortunately, the rapid

J. Biswas is currently affiliated with Hewlett Packard Enterprise, USA.
This research was supported by NSF Grant No. CNS-2011252 and an Intel grant.
Authors' addresses: J. Gupta, K. Kant, and J. Biswas, Computer and Information Sciences, Temple University, 1925 N. 12th Street, Philadelphia, Pennsylvania, USA, 19122; e-mails: jit.gupta@temple.edu, kkant@temple.edu, joyantamishu@gmail.com; A. Pal, Computer Science and Engineering, Indian Institute of Technology Kanpur, Kanpur, India; e-mail: amitangshu.pal@gmail.com.

2376-3639/2024/01-ART4 $15.00
https://doi.org/10.1145/3631606

evolution in storage technologies has made it possible to continuously improve the size, speed, cost, and flexibility of storage systems. In particular, mechanical **hard-disk drives (HDDs)** have given way to much faster **solid-state disks (SSDs)** for storing much of the popular data, while both technologies continue to evolve rapidly. There are even faster storage technologies whose latencies approach the DRAM memory, and this has led to the notion of ***storage class memory* (SCM)**, which refers to the combined features of nonvolatility (persistence) and speeds approaching that of memory. Thus, SCM can truly fill the gap between storage and memory. For example, a fast SCM could be organized and accessed just like DRAM, meaning that the CPU directly accesses the SCM in small units (a few cachelines at a time) and waits for the access to complete. It can also be organized as storage, where the transfer sizes are large (e.g., 4 KB or larger) and the CPU does not wait for IO completion. Fast SCM accessed like a memory (henceforth called ***persistent memory*** or **PM**) is attractive because of its expected lower cost compared to DRAM.

1.1 Data Center Storage Organization

Storage systems are invariably organized as separate subsystems, known as storage servers, each hosting many storage devices. The storage server manages these devices and provides the ability to allocate storage and provide access to them without regard to how the allocated space is distributed over various devices. Thus, all storage accesses from any application go over the network via network switches to the storage servers, as shown in Figure 1(a). The emerging PM, which provides nonvolatile storage with latencies of about an order of magnitude slower than DRAM (but much faster than storage) can also benefit from being installed in a storage server and accessed over the network. Although latency considerations would suggest that the PM be deployed on each application server locally, the shared remote access not only allows on-demand access to PM but also incurs overall lower cost. However, the storage and PM accesses in such a case would share the same network and thus require appropriate QoS so that a low latency can be maintained for PM even when the storage accesses experience congestion.

It is important to note that modern storage devices accessed over the network are fast enough for the network latency to become a significant part of the overall access latency, which was not an issue for the older HDD-based storage. The oversized impact of network latency is also facilitated by highly efficient storage access protocols such as the **Non-volatile Memory Express (NVMe)** protocol [9]. As shown in Figure 1(b), when accessing an SSD locally, the access latency is of the range of 30–100 μs; however, a TCP-based network going through multiple switches can add substantially to this latency even without network congestion. Furthermore, with current SSDs already exceeding 1M ops/s (or 32 Gb/s), *a few SSDs can easily congest even a 100 Gb/s link*. This has two implications: (1) Network congestion could occur rather frequently, and (2) storage-side congestion is less likely, since the increasing storage demands would generally require many more SSDs to the storage server than what the network can handle at peak transfer rate.

1.2 End-to-end QoS Differentiation

With increasing demands of large amounts of data to be processed online, it is important to differentiate between various applications in terms of their storage access latency and throughput. While the applications may have specific high-level **Service-level Agreements (SLAs)**, it is difficult to translate those directly into QoS differentiation, since enforcing it involves many factors beyond network bandwidth, and includes computing resources, storage side hierarchy through which the data flows, host side caching, and so on. Thus, we consider QoS differentiation only in terms of relative treatment of different classes of applications (e.g., relative throughputs). We will assume that the applications have already been classified into a small number of classes, each of which may be further designated as latency or throughput-sensitive. We then specify the relative

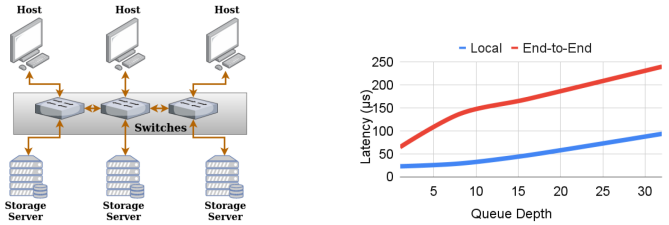


Fig. 1. (a) End-to-end data center organization and (b) comparison between local and end-to-end storage access.

treatment in terms of the four key components on the storage path: host-side, network, storage protocol, and the device. The SLA for the latency-sensitive application class often specifies some bounds on the tail latency (e.g., 99 percentile latency), but latencies can be controlled directly only for classes that are given absolute priority over all others and their offered rate remains low enough that they can always be accommodated at the cost of others. Beyond this, while it is possible to enforce relative latencies (instead of relative throughputs), this is meaningful only for brief congestion, since the queue lengths will continue to increase under sustained congestion. The relative throughput control does work under sustained congestion and would also impact the latencies under transient episodes based on the difference between offered and carried load for each class.

One issue concerning the enforcement of relative throughputs is the coordination across multiple resources in the path. By using the same relative throughputs, we can ensure that the resource that is most bottlenecked will automatically determine the end-to-end throughput of each class without requiring further coordination. However, enforcing the same ratios end to end does require a tagging mechanism so that the treatment is conveyed and followed consistently.

1.3 Our Contributions

We make the following contributions towards **end-to-end (E2E)** QoS for modern storage systems, where our focus is on the *differentiation* (or relative treatment) of different classes of activities rather than guaranteed performance.

- **A.** We propose a mechanism to introduce two QoS-aware transport protocols called QTCP and QRDMA that introduce QoS differentiation into existing base protocols called DCTCP and DCQCN, respectively. We evaluate them for both throughput-sensitive and latency-sensitive storage traffic. We show that they cover a much more general scenario and consistently provide stable and desired differentiation in all cases. We also propose a discrete-time analytic model for both and demonstrate agreement with the experimental results.
- **B.** We consider the coexistence of QTCP (used for storage) and QRDMA (used for PM) to emulate mixed PM/storage network traffic and show how we can protect the QRDMA PM traffic from QTCP traffic and achieve PM latency limited only by the unavoidable **head-of-the-line (HoL)** blocking on the storage receive end that accepts both storage and PM requests. To the best of our knowledge, this is the first detailed examination of mixed PM/storage network traffic.
- **C.** We integrate a comprehensive storage device model (consisting of SSD internals), access protocol (i.e., NVMe) model, and the detailed implementation of QTCP/QRDMA to enable experimentation with the implementation of E2E QoS differentiation. We do this in the simulation domain for several reasons: the ability to modify SSD behavior (which is impossible in reality, since all SSD internals are invariably proprietary), avoiding the difficulties associated

Table 1. Common Abbreviations

SNIA	Storage Networking Industry Assoc.	HDD/SSD	Hard-disk/Solid-state Drive
FTL	Flash Translation Layer	SLC/MLC	Single-/Multi-level (2) Cell Flash
TLC/QLC	Triple/Quad Level Cell Flash	SCM	Storage Class Memory
PM	Persistent Memory	SQ/CQ	Submission/Completion Queue
LBA	Logical Block Address	WRR	Weighted Round Robin
PFC	Priority Flow Control	NVMe	Nonvolatile Memory Express Interface
NVMe-oF	NVMe over Fabric Interface	ECN	Explicit Congestion Notification
CNP	Congestion Notification Packet	HOL	Head of the Line
E2E	End-to-end	QoS	Quality of Service

with kernel programming needed to modify network protocols, and the ability to change things freely. Our models do make use of available software packages that are extremely comprehensive.

D. By using the integrated model, we explore the consequences of having or not having QoS treatment in various places and demonstrate when coordinated QoS is needed and how it affects performance. To the best of our knowledge, this is the first such exploration and it provides important insights into how E2E QoS should be configured for networked storage.

The rest of the article is organized as follows. Section 2 discusses the background and motivation of our work coupled with the limitations of existing works, while Section 3 discusses our proposed mechanisms. We model our proposed technique in Section 3.1.2, followed by extensive evaluation in Section 4. We additionally discuss some more works of interest and their limitations in Section 5. The article is concluded in Section 6. Table 1 includes some of the key abbreviations used in this article.

2 BACKGROUND AND MOTIVATION

In this section, we discuss the necessary background relating to the host-to-device storage access path. There are four elements in this path: host-side storage stack, storage access protocol, network transport for storage access protocol, and storage device itself. In each case, we point out the QoS features that are available or in need of development. This discussion would then motivate the discussion in the rest of the article about mechanisms that we have either invented or harnessed to configure a consistent E2E QoS treatment of storage request of various sorts.

2.1 Host-level Storage Stack

The host side of storage interface is quite complex and involves multiple layers. In most cases, the application's interaction with storage is through a file-system that the host has defined over the storage space allocated for it. The file system maps the IO requests to the allocated storage, which is viewed simply as a sequence of "logical blocks." These are typically 4 KB chunks and addressed using *logical block addresses* or LBAs. The block layer then interacts with the NVMe layer to submit the LBA I/O requests and retrieve completions. For efficiency, the host maintains a "page cache" in the DRAM that holds the recently accessed and popular blocks (or pages). In general, the page cache is quite large and helps reduce IO latency substantially, in addition to allowing the real IO to be done in much larger chunk sizes (e.g., default of 16 KB for Linux). In terms of QoS, while it is possible to give differentiated treatment to requests at multiple points in the host stack, the most consequential is the page cache. In particular, an I/O request for a higher class may be allowed to evict a chunk of lower QoS class to make room for the new read/write. While there are many issues in how to do this well, these are all classical caching issues that we have chosen not to investigate here. Instead, our focus is on *E2E treatment of IO requests that result in a device access.*

2.2 High-performance Storage Interfacing

Given the need to serve very high-throughput and low-latency storage devices, the storage access protocols themselves need to be high performance. The **NVMe (non-volatile memory express)** protocol [29] is one such complex protocol that supports many features relevant to SSDs, including a differentiated queuing structure later shown in Figure 4. There are five queuing levels (admin, urgent, high, medium, low), where the (single) "admin" queue is reserved for administrative commands (highest priority), the "urgent" level is reserved for latency-centric IO operations that get strict priority over others, and the remaining three are for throughput-centric traffic. The last 3 queues operate according to **weighted round-robin (WRR)** scheduling, with high, medium, and low having decreasing weight. Each level could have multiple submission queues, served in a round-robin manner. We make use of this queuing structure in our exploration. In particular, for the mixed situation involving both storage and PM, we use the "urgent" queue for PM requests. It is important to note that the NVMe queuing differentiation does not dictate anything to the device itself—it is only used by the device to determine how many requests to remove from each queue and in what order, upon each IO completion. How the device processes them is up to the device.

Extending the storage access over the network requires the local access protocol such as the NVMe to be transported from the host to the device without changing the basic access semantics. **NVMe over Fabric (NVMe-oF)** [30] provides this capability by encapsulating the submission and completion queue entries into transport-independent "capsules" for transfer between the host and the storage device. The capsule transport also puts the command queue in the memory of the device controller, known as the **Controller Memory Buffer (CMB)**, rather than the host memory, thereby further improving latency. The completion queue stays with the host.

NVMe-oF is intended to run on top of an E2E network transport so that the storage server can be located anywhere on the data center network and still be accessible to all the hosts. To maintain high throughput, the transport must be not only reliable but also loss-free, since the management of packet loss causes substantial disruptions in the delivery of data and hence stalls for the applications. Thus, the clear choices for appropriate transport are lossless versions of TCP and a lossless implementation of RDMA over routable Ethernet network (i.e., RoCEv2, which implements RDMA protocol on top of IP layer). For data center use, there are two widely used [5, 42] protocols that we shall use as baselines. One is the so-called ***Data Center TCP*** **(DCTCP)** [2] and the other is ***Data Center Quantized Congestion Notification*** **(DCQCN)** [48]. The latter builds a lossless RDMA service on top of RoCEv2 by using some data center Ethernet features as described later.

2.3 Characteristics of Emerging Storage Technologies

The dominant storage devices in modern systems are SSDs, which are based on the "flash" technology. The key characteristic of this technology is that it does not allow overwrites. Thus, a "cell" of a flash (which can store 1, 2, 3, or 4 bits depending on technology) must first undergo a rather slow erase operation before its stored value can be changed. Frequent erases are avoided by writing the updated data elsewhere until the fraction of fresh cells falls below some threshold. Managing such out-of-place writes introduces significant complexity including (a) the need to distinguish between the logical address that is apparently written to and the real (or physical) address where the write occurs, and (b) creation of "garbage" in form of obsolete data, which must be periodically collected, those cells erased, and returned to the fresh pool. Furthermore, a cell can only be written a certain number of times (known as its *endurance limit*) before it wears out, and it is thus important to spread the write out over all the cells so that they age similarly. Yet another complexity comes from the organization of the cells. Cells are grouped into "pages," usually of size 4–16 KB, and all reads/writes occur at this granularity. Figure 2 shows backend details of the SSD where the

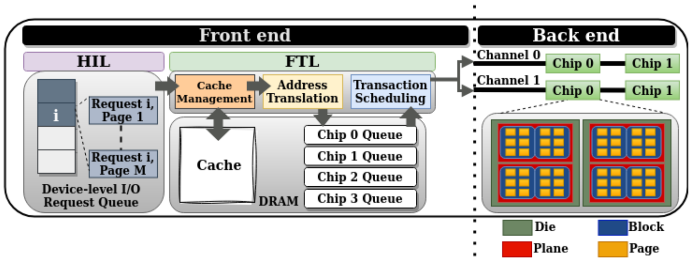


Fig. 2. Internal architecture of an SSD.

aforementioned pages are further grouped into "blocks" (or flash-blocks, different from storage system notion of a block), of sizes 128 KB–8 MB. Erasures must occur at the level of flash-blocks, which introduces further complexity. The flash-blocks are further grouped into "planes," and an erasure generally blocks the entire plane. Multiple planes form a chip-die, and a physical flash-chip is divided into multiple dies. Each chip connects to the CPU/memory hub over one or more channels for data transfer. The lower-level structures (dies and planes) are also connected via internal busses that often become bottlenecks.

These complex management tasks are carried out by the built-in firmware of the SSD known as the **Flash Translation Layer (FTL)**. FTL is generally vendor-specific and proprietary, although open-source versions such as OpenSSD [31] do exist. Figure 2's frontend shows the FTL interfaces with the NVMe controller via a **Host Interface Layer (HIL)**. The HIL transfers requests from the storage interface queues (e.g., the NVMe submission queues explained in Section 2.2) to a device-level I/O buffer and then passes on the requests to device queue managed by the FTL. Since the FTL performs read/write at the granularity of a page, larger transfer requests are broken into "transactions," each of size one page, before entering them into SSDs queues. Most implementations have no notion of QoS at the device level, although a few rather weak mechanisms have been proposed and implemented in a few SSDs. The best known of these is the notion of "write streaming" [23, 24, 34], wherein the host provides hints to the FTL about the life-time of the flash-blocks, so they can be handled more efficiently. This is neither a QoS mechanism nor easy to implement reliably and thus has not received much acceptance.

One consequence of all the complexity is that the SSD tail latency can stretch into several milliseconds, for example, when an IO is delayed by a long running garbage collection.

In addition to flash, there are numerous other storage technologies that are in various stages of development and commercialization [1], mostly with much lower latency and higher endurance than flash. The commercially available ones include Intel's Optane technology[1] and Kioxia's XL-flash technology. These technologies are appropriate for use both as normal storage (with access latencies in the range 10–20 μs [45]) and also as *PM* with substantially lower access latencies, typically less than 1 μs. On the flash side, even the rather inexpensive SSDs available today are capable of driving up to 25–35 Gb/s IO throughput [10] with nominal latencies of under 100 μs [38].

With storage semantics, the PM access sizes will typically be large (e.g., 4 KB), and the thread requesting PM access will be switched out until the response is received. With the memory model, the access sizes are typically a few cachelines (e.g., 128 or 256 bytes) and the CPU will stall until the data is received. This makes the PM access latency crucial. A substantial network delay on the path would render the memory model worthless. This is particularly true if the PM traffic has

[1]Recently Intel discontinued further development of Optane technology for business reasons, but has also released latest SSDs using this technology.

to compete with storage traffic through the network. Thus, providing an urgent QoS to PM traffic becomes essential. Assuming that the PM traffic forms only a small portion of the overall bottleneck link capacity, it is desirable to not throttle the PM traffic at all during congestion. Furthermore, the switches (in addition to the endpoints) should also provide preference to the PM traffic. We will show that such network-level intervention can reduce the PM latency substantially.[2]

2.4 Network Issues for Remote Storage Access

As stated earlier, the high throughput of modern storage devices can easily congest even high-speed 100 Gb/s Ethernet links. Thus, managing network congestion and providing differentiated treatment under network congestion scenarios becomes a crucial part of the transport network for storage access. We shall use the previously mentioned DCTCP and DCQCN as the baseline transport protocols to build the QoS features, because these protocols are published as IETF RFCs [5] and available in some switch and NIC implementations[42].

Both DCTCP and DCQCN depend on the notification of congestion to the sending endpoint via standard mechanisms. In case of DCTCP, it is the ***explicit congestion notification*** **(ECN)** and involves two bits in the packet header. One is set when any switch on the forward path experiences congestion, defined as the queue length in the switch exceeding some predefined threshold. When the packet reaches the receiver, it sets the second bit (the "echo" bit) to inform the sender of congestion via TCP ACKs. For low latency, DCQCN is implemented on top of UDP (rather than TCP) and thus does not have any ACKs. DCQCN still uses the ECN bit for marking packets at congested switches, but instead of an ACK, the receiver sends an explicit **congestion notification packet (CNP)** to the sender. DCQCN also utilizes the data center Ethernet QoS feature called **PFC (priority flow control)** for selectively pausing the flow towards a congested destination. The standard PFC mechanism operates at layer 2 and pauses the flows to all transport destinations on the link. DCQCN tackles this issue by leveraging both the PFC and ECN mechanism.

As such neither DCTCP nor DCQCN are QoS aware, but we demonstrate how their dynamics can be changed to make them QoS-aware. A key reason to consider DCQCN-based protocol is to provide low latency for the small transfers needed for remote PM access. However, when PM and storage traffic share the same network, it is also important to give preferred treatment to PM traffic at the switches as well. This is because the switch buffer is shared amongst multiple outgoing ports and this could lead to HoL blocking. We shall propose a simple priority queuing mechanism for this that can be implemented easily but does require the switches to recognize the PM traffic.

It may be noted that Ethernet networks do provide QoS features at layers 2 and 3, but they are not useful for our purposes. In particular, the IP layer provides a standard mechanism called DSCP (differentiated services code point), but was defined for wide area networks, with packet loss-centric QoS treatment. Furthermore, DSCP defines a "hop-by-hop" QoS treatment (in each router), rather than a uniform E2E treatment. Thus, DSCP, as defined, is not useful for QoS differentiation. We later on also discuss other QoS aware solutions (e.g., HOMA, PDQ, L^{2}DCT, D^{2}TCP, and D3) along with their limitations in Section 5.2.

3 E2E QOS DIFFERENTIATION IN NETWORKED STORAGE TRANSFERS

In this section, we divide our E2E QoS discussion into three major parts. We first talk about our proposed QTCP and QRDMA modifications in the network along with meeting QoS requirements for ultra low-latency traffic. Following this, we discuss how we achieve QoS differentiation in the

[2]All of these techniques may not provide low enough latency to make the memory model feasible, in which case, it may need to use the storage model but with small transfers.

storage end. Finally, we explore appropriate mechanisms to mark the network packets for E2E configuration of QoS.

3.1 QoS Differentiation in the Network

3.1.1 QoS Aware DCTCP and DCQCN (QTCP and QRDMA). We first define how the QoS metrics are utilized by our proposed QTCP and QRDMA mechanisms in the network during congestion. The QoS differentiation is achieved by enforcing a relative ratio of either throughputs or latencies depending on the objective (throughput or latency). We generally do not mix the two, since that would result in undesired interactions; however, there could be one case, where the combined treatment is meaningful: All latency-sensitive classes are given strict priority over others, and then relative latencies are enforced for high priority classes, and relative throughput for low priority classes. We assume that the bottleneck bandwidth is known as it can be estimated from the techniques used in Reference [7], while *the target bandwidth is a predefined fraction of the available bandwidth (depending on its bandwidth objective).* For latency differentiation, we consider the tail-latency objective to be defined by the same percentile value, e.g., 90th percentile latency of 100 μs,[3] while for throughput differentiation, we consider the minimum bandwidth requirement of a flow. As mentioned in Section 2.4, the ECN mechanism is used to detect congestion.

For our proposed mechanisms, we introduce a new QoS class-specific measure called Quality Factor, denoted as Q_i for a class i. It is defined as the ratio between the target bandwidth and the actual available bandwidth of that class, where the available measured bandwidth is exponentially smoothed over. Hence, Q_i would then be defined as

$$Q_i = \lambda_{it}/\lambda'_{ia} \text{ where } \lambda'_{ia} = (1-\gamma)\lambda_{ia} + \gamma\lambda'_{ia}, i = 1..K, \tag{1}$$

where λ'_{ia} and λ_{it} denote, respectively, the actual smoothed bandwidth and the target bandwidth, while γ is the smoothing factor. Q_i is a unit-less metric as it is a ratio. A value of $Q_i < 1$ denotes that the flow in question has slack available, meaning that its corresponding window can be squeezed or reduced to make way for other flows. $Q_i > 1$ denotes that a deficit has been detected and the flow's window needs to be increased to meet its QoS requirements. The flow rate for each flow is modulated separately as it is assumed that flows have separate connections.

We can similarly define Q_i for tail latency too. However, in this case, the ratio is reversed to preserve the understanding that a value of $Q_i < 1$ denotes slack and $Q_i > 1$ denotes deficit as before. So in this case, Q_i looks as follows:

$$Q_i = L'_{ia}/L_{it} \text{ where } L'_{ia} = (1-\gamma)L_{ia} + \gamma L'_{ia}, i = 1..K. \tag{2}$$

Similar to DCTCP, the quantity of interest is the fraction of ACKs or CNPs received in an RTT window that indicates congestion (i.e., ECN marked). This fraction can be monitored separately for each flow with a unique QoS class, and we henceforth denote it as $f_i(n)$ for ith QoS class and the nth RTT window. $f_i(n)$ can be rather unstable, and thus we use exponential smoothing over it, exactly as in DCTCP/DCQCN. That is, if $\alpha_i(n)$ denotes the smoothed congestion fraction of class i at RTT n, then we have

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

where $0 < g < 1$ is the smoothing constant, and by default is set as 0.5. It may be noted here that in DCQCN, at most 1 explicit CNP is sent when there is congestion detected in the last interval, and thus $f_i(n-1)$ can only be 0 or 1.

DCTCP reduces the window per **round-trip time (RTT)** in proportion to the latest estimate of α_i such that in the limiting case of $\alpha_i = 1$, the window is halved. Hence, the window controlling

[3]In case the percentile value is different, we can attempt to estimate it using known methods such as Chebychev's inequality.

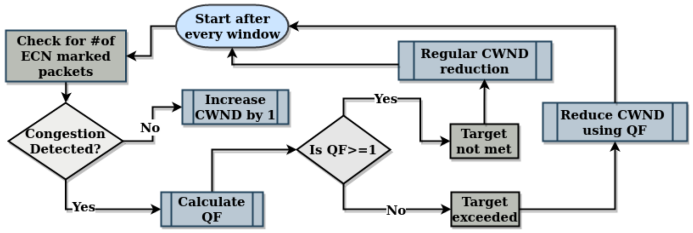


Fig. 3. Flowchart for Quality Factor (QF)-based flow control.

mechanism is formulated as follows:

$$W_i(n) = W_i(n-1)\left(1 - \frac{\alpha_i}{2}\right). \tag{4}$$

Similar to DCTCP and DCQCN, 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 for the different flows. We term this modified version of DCTCP as QTCP and redefine the window size W_i for a flow i as follows:

$$W_i(n) = \begin{cases} W_i(n-1) + 1, & \text{No ECN,} \\ W_i(n-1)(1 - \frac{\alpha_i}{2}), & \alpha_i \geq 0 \text{ and } Q_i \geq 1, \\ W_i(n-1)(1 - \frac{\alpha_i}{2})Q_i, & \alpha_i \geq 0 \text{ and } Q_i < 1. \end{cases} \tag{5}$$

In fact, if $Q_i = 1$, then the above equation reduces exactly to what is used by DCTCP/DCQCN. However, in that case, no QoS distinction is possible as shown in our results later.

The above is applicable to every update interval, i.e., RTT. As shown in Figure 3, at every interval QTCP checks for the fraction of ECN marked ACKs received, which signifies congestion. It refrains from using the quality factor metric when there is no congestion and even when congestion is encountered, but deficit is detected by the quality factor estimation (i.e., $Q_i \geq 1$). The latter scenario demonstrates that QoS requirement has not yet been met for the flow and so only the value of α_i is used for flow rate modulation. However, when a slack is detected, i.e., $Q_i < 1$, the QoS requirement has already been met and so the flow should be backing off to free resources for the other competing flows whose QoS requirements have not been met.

For RDMA context, however, there is no concept of RTT as it is a connectionless protocol. Hence, similar to DCQCN, which updates the flow rate in every N microsecond intervals, our proposed QRDMA also updates the flow rate in the same manner. The updated equations look as follows where RC_i is the current flow rate:

$$RC_i(n) = \begin{cases} RC_i(n-1), & \text{No ECN,} \\ RC_i(n-1)(1 - \frac{\alpha_i}{2}), & \alpha_i \geq 0 \text{ and } Q_i \geq 1, \\ RC_i(n-1)(1 - \frac{\alpha_i}{2})Q_i, & \alpha_i \geq 0 \text{ and } Q_i < 1. \end{cases} \tag{6}$$

Similar to traditional DCQCN, QRDMA also stores the current flow rate RC_i as the target flow rate RT_i as soon as congestion is detected. This is to aid the fast recovery and additive increase for flow rate increase (when congestion period is over).

3.1.2 Analytical Modeling of QTCP and QRDMA. We now model our proposed mechanisms QTCP and QRDMA. We consider a ***Discrete Time Model (DTM)*** where the flow rate changes in every update interval. Another possibility is the ***Fluid Flow Model (FFM)*** similar to the one used for the analysis of DCTCP [4]. However, a fluid model assumes a continuous and incremental change in all parameters including the fraction of marked packets, queue length, and the window size. We believe that a discrete model can capture the dynamics better and is simpler, since

Table 2. Description of Important Terms Used in Our Analysis

C_i	Allocated bandwidth to flow i	R_i	Round Trip Time for flow i
W_i	Window size for flow i in number of packets	Q_i	Quality Factor for flow i
r_i	Ratio of bandwidth for flow i	q_{ai}	Queue length observed by packet of flow i
d_i	Baseline latency	α	Fraction of ECN marked packets

it needs difference equations instead of differential equations. The simplicity is helpful, since, unlike DCTCP, we now need to write equations for each class, which results in a coupled system of equations. Note that both FFM and DTM have their weaknesses, since neither captures the precise behavior of the system in terms of when things are updated. However, we show that DTM provides very accurate results in all cases.

The current update slot for the DTM is termed as n, i.e., the current time slot and it is assumed that there are $i = 1..I$ active connections, each of which belongs to a specific QoS class. The terms class and connections are used interchangeably. If we consider the total capacity of a bottleneck link to be C, then the share allocated to a class i during an update slot n can be denoted as $C_i(n)$, such that $\sum_{i=1}^{I} C_i = C$. The queue length of class j as observed by an incoming class i packet at the *bottleneck egress port of the switch* is denoted as $q_{aj}^{(i)}(n)$ at an update interval n while the RTT is denoted as $R_i(n)$. We also need to take into consideration the event that represents the state where the switch queue buffer is at or beyond the threshold K. This is signified by $e_i(n)$ where i denotes the class whose packet observes this event, thus resulting in its CE (congestion experienced) bit being set. In the DTM, this event concerns the observed situation in the previous time slot. That is,

$$e_i(n) = \mathbb{I}_{\sum_{j=1}^{I} q_{aj}^{(i)}(n-1) \geq K}, \tag{7}$$

where $\mathbb{I}$ is the indicator function (1 if the condition is true, and 0 otherwise). Even though the observed distribution of packets in the queue may be different for different incoming classes i, we assume that the switch marks packets of all classes uniformly once it crosses the threshold K. Hence, $q_{aj}^{(i)}(n)$ has a weak if not negligible dependence on this. Thus, we can henceforth denote this event (i.e., congestion detected) as just $e(n)$. For our DTM, we also need not observe every single event but just the probability $p(n)$ that the queue is full. Which can be approximated as follows:

$$p(n) = \begin{cases} 1 - \frac{K-1}{B(n-1)} & \text{if } B(n-1) \geq K, \\ 0 & \text{if } B(n-1) < K, \end{cases} \tag{8}$$

where $B(n-1) = \sum_{i=1}^{I} q_{ai}(n-1)$. We now estimate the latency $L_{ai}(n)$ experienced by a class i packet. It can be gauged as $L_{ai}(n) = d_i + q_{ai}(n)/C_i(n)$ where the baseline latency is d_i, which encompasses all delays that are independent of queuing delay, for example, send/receive processing delay at the endpoints, transmission, and propagation delay, and switch processing delay. It is assumed that the feedback packets face negligible queuing delay, and the basic delays are symmetric in the two directions. That is, $R_i(n) = d_i$. With QTCP, the feedback uses ACKs, which will be largely implicit if the backward traffic rate is high, but may involve explicit ACKs whenever there are gaps in the backward traffic. In either case, the queuing delay experienced by the feedback will be negligible. With QRDMA, the ACKs are necessarily explicit but must be transmitted at the highest priority for it to work properly, and thus should experience negligible queuing delay. Note that the latency here does not include any application-level latency, which could grow until a timeout, user abandonment, or service denial via admission control occurs.

Table 2 consists of significant terms and their descriptions used in this analysis. We now consider the two metrics of significance to us.

Throughput: Every class i is allocated a given ratio of bandwidth r_i, relative to r_1, i.e., the bandwidth of class 1, which is unity. Hence, independent of the update slot, $C_i = C.r_i/\sum_i r_i$. We can safely assume that no packets are dropped from our previous discussion and hence the experienced throughput can be estimated from the number of transmitted packets $W_i(t-\overline{R_i})$ during the preceding update slot $R_i(t-\overline{R_i})$, i.e., $\lambda_i(n) = W_i(t-\overline{R_i})/R_i(t-\overline{R_i})$. Hence, the quality factor for a class i can be described as $Q_i(n) = C_i/\lambda_i(n)$

Latency: As discussed before, only queuing latency is of importance for our DTM. It is assumed that the QoS classes are ordered according to a score, such that class 1 is the most significant. The target latency L_{it} for all the classes needs to be large enough to ensure that they are attainable. It is to be remembered that the switches use **First Come First Serve (FCFS)** treatment for all incoming packets.[4] Hence, the latency can be described as $L_i(n) = d + q_{ai}(t-\overline{R_i})/C_i(t-\overline{R_i})$, where $C_i(n) = W_i(n)/R_i(n)$. From this, we can estimate the Quality Factor $Q_i(n)$ as $Q_i(n) = L_i(n)/L_{it}$. From our discussion till now, we can rewrite our equations as follows:

$$C_i(n) = \begin{cases} C.r_i/\sum_i r_i & \text{Throughput control,} \\ \frac{W_i(n-1)}{R_i(n-1)} & \text{Latency Control,} \end{cases} \tag{9}$$

$$Q_i(n) = \begin{cases} \frac{C_i(n)R_i(n-1)}{W_i(n-1)} & \text{Throughput control,} \\ \frac{d_i + q_{ai}(n-1)/C_i(n-1)}{L_{it}} & \text{Latency Control,} \end{cases} \tag{10}$$

$$\alpha_i(n) = \alpha_i(n-1) + \gamma[p(n) - \alpha_i(n-1)]. \tag{11}$$

Note that all units are in terms of packets and not bytes. We first estimate the probability $p(n)$, i.e., the event for the queue being full, by observing if the ECN marked packet has been received depending on the queue length during the previous update slot($q_{ai}(n-1)$). The value of $p(n)$ is used to determine the ratio of bandwidth allocated to a class during the current time slot, i.e., $C_i(n)$, which in turn is used to compute the Quality Factor $Q_i(n)$. Following this, we update the value of α, which helps in calculating all the number of packets that are transferred, i.e., $W_i(n)$ and the update interval ($R_i(n)$) during the current time slot. Both $W_i(n)$ and $R_i(n)$ are then in turn used to estimate the queue length $q_{ai}(n)$ for the current slot and the process follows for succeeding slots to keep the evolution ongoing. Hence,

$$W_i(n) = \begin{cases} W_i + 1 & \alpha_i(n) \le \varepsilon, \\ W_i\left[1 - \frac{\alpha_i(n)}{2}\right] & \alpha_i(n) > \varepsilon \& Q_i(n) > 1, \\ W_i\left[1 - \frac{\alpha_i(n)}{2}\right] Q_i(n) & \alpha_i(n) > \varepsilon \& Q_i(n) \le 1, \end{cases} \tag{12}$$

$$R_i(n) = 2d_i + B/C, \tag{13}$$

$$q_{ai}(n) = \max[0, q_{ai} + W_i(n) - C_i R_i(n)]. \tag{14}$$

It is difficult to seek the steady state for this system when W_i and R_i change (as seen in Equation (12)), but it could be sought for the remaining cases when their values do not change from one slot to another. We assume in our proposed equations that are always packets to fill up the estimated W_i during each slot. A packet generation process can be included to extend the proposed DTM and also by keeping track of the packets for class i that have not been transmitted yet—$U_i(n)$. The modified window size for class i, termed as $W_i'(n)$, can then be represented as the

[4]Multi-class open-system queuing formulae [20] can be used to gauge the range of usable values.

minimum of the estimated window size $W_i(n)$ (from our previous equations) and $U_i(n)$. Hence,

$$M = \inf_{\left(\sum_{m=1}^{M'} G_i^{(m)}\right) > R(n-1)} (M'), \tag{15}$$

$$U_i(n) = U_i(n-1) - W_i'(n-1) + M - 1, \tag{16}$$

$$W'(n) = \min[W(n), U_i(n)], \tag{17}$$

where $G_i^{(m)}$ is the time elapsed between succeeding packets $(m-1)$ and m during an RTT slot. This time measure is governed by the arrival distribution of the packet, which could be bursty in nature.

3.1.3 QoS Differentiation for Mixed Traffic. We now look into the mixed network traffic that comprises of both storage traffic (with transfer sizes or 1 or more LBAs) and ultra low-latency traffic (e.g., PM traffic with transfer sizes of only 2–4 cachelines).

Providing very low latency to a specific class of network traffic requires that such traffic gets the highest priority in the network switches as well, henceforth called "urgent" priority, following NVMe terminology. Since the data transfer latencies for storage traffic are rather high and storage access model is less affected by latency than a memory access model, no in-network priority is essential for storage traffic. Both types of traffic do need differentiation at network endpoints in all cases. Furthermore, as already stated, the host, network, NVMe, and the device must apply consistent QoS treatment, which in turn requires an E2E communication of priority classes.

In this work, we use the urgent priority for ultra-low-latency traffic in two ways: (a) Ample space reservation in the egress switch port, and (b) Giving strict (but non-preemptive) priority to the Urgent class queue over other queues. In summary, if the buffer size is B packets, then the best QoS treatment for Urgent traffic can be achieved if there is a buffer reservation P for it. This traffic (using QRDMA as a transport) will still have to wait for the ongoing transmission from the other queue, however, the added delay is at most one packet transmission time (120 ns at 100 Gb/s). A larger HoL blocking occurs on the receive end, since the receiver will receive packets strictly in the order they arrive on the receive side. The egress port buffer would be designed so that it is normally not overrun and thus the described congestion control mechanism for QRDMA does not kick in. However, it is not possible to guarantee that the Urgent traffic is not throttled unless the host uses a suitable admission control mechanism to ensure that this traffic does not exceed the design limits. We utilize this prioritization technique for our evaluation of PM traffic in Section 4.4.

3.2 QoS Differentiation at the Storage End

For E2E QoS differentiation, we also need to ensure that the storage end provides QoS differentiation for incoming flows. As mentioned earlier, the storage end itself is comprised of two components—the interfacing device access protocol (i.e., NVMe) and the storage device (i.e., the SSD). The NVMe WRR queue arbitration mechanism can help in throughput differentiation. For example, if the weights for the queues are in the ratio of 3:2:1, then the SSD receives two Medium class requests after every three High class requests. However, how these requests are scheduled and serviced inside the actual device is a more complex problem (due to the issues mentioned in Section 2.3). Hence, access latencies can be non-deterministic. It is also to be noted that requests can be of varying sizes and thus this is not enough to guarantee latency differentiation. We have extended the WRR principle to the in-device queues too as shown in Figure 4. We show that even though WRR is used to divide throughput in a fixed ratio (e.g., 3:2:1), it can be effective in providing latency differentiation too. Also, chip-level queues schedule requests in the size of single pages. This gives us more control on the latencies. Thus, we show how this throughput differentiation

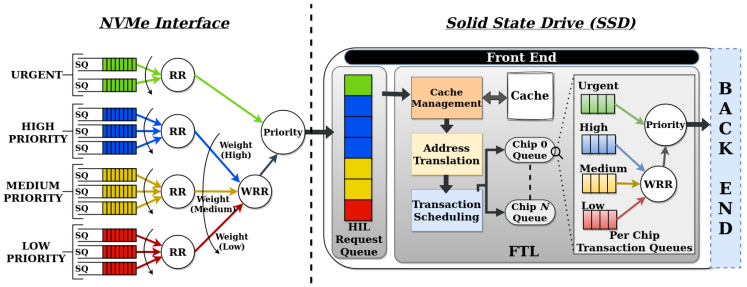


Fig. 4. Extending NVMe WRR inside the SSD.

mechanism can also be used to provide treatment for latency, although not in the exact WRR ratios (as shown later in Section 4.2).

Additionally, the NVMe protocol has introduced the notion of "predictable-latency mode," which allows background operations to be confined to certain periods known as **"nondeterministic" (NDT)** periods. We have explored this extensively in References [36, 37] to provide deterministic latency to different applications. We introduced a module called PLMC (i.e., PLM Controller) that allocates the number of "deterministic" accesses an application can perform based on its QoS class. This could also be used in conjunction with our proposed NVMe WRR extension to tackle the large latency values caused by background activities. However, that is beyond the scope of this work.

Finally, we have not considered the low-probability scenario where bursty traffic may lead to request dropping at the storage end, since this would be very disruptive to the application. Application threads attempting to do a read will typically stall until the data is returned, and long delays and inter-dependencies (between requests) would tend to slow down and ultimately stop request generation from the application itself. However, data generation type of applications could continue to pump more data and would ultimately need to be prevented from generating new write requests using some admission control mechanism. This could, in effect, result in data loss if the data comes from some physical system that cannot be stopped, but the storage management system should not explicitly drop requests.

3.3 End-to-end Configuration of QoS

E2E QoS requires a consistent treatment of QoS in the network, at NVMe level, and in the device. NVMe already provides a rich QoS classification, and thus we can leverage the available mechanisms. The storage side has no standardized or universal notion of QoS for SSDs, although several mechanisms have been proposed in the literature and some others implemented by specific vendors in their products. We have configured a mechanism similar to NVMe for the SSDs as well, namely, four priority queues including an "urgent" queue and three others with WRR scheduling called High-, Medium-, and Low-priority queues (where a high priority queue has larger WRR weight). This priority queuing can be at multiple levels within the SSD, the most relevant ones being the (a) chip level (or coarse grain), and (b) Plane level (Fine grain). The chip-level differentiation would account for priority treatment considering the rather limited bandwidth of internal buses in real SSDs, but it is rather coarse, since the SSD typically has only 2–16 chips. Queuing at the plane level becomes rather fine-grain, since a chip might have many planes (128–1,024) but would not add much value unless except in rather pathological cases of hot spots within each plane. For simplicity, we have configured it only at the Chip level. It is possible study impact of variable granularity with this configuration by varying the number of chips and planes/chip, but that level of detail is outside the scope of this article.

The four QoS classes discussed so far (three QoS classes for storage traffic and one for PM) need to be carried from host all the way to the device for each storage read/write operation to support E2E QoS. Furthermore, each layer along the way should be able to access it without much overhead to provide the desired QoS treatment. While conceptually straightforward, providing this capability in real storage systems involves the rather cumbersome issues of protocol changes and standardization, both of which are outside the scope of this exploratory study. Nevertheless, for the sake of completeness, we briefly discuss how an E2E QoS hinting can be supported in contemporary storage stacks.

With four QoS classes, we need to find at least two bits in the headers read and write commands to convey the QoS class. NVMe read/write commands define a few bits in Dword (double word) #13 that relate to latency and frequency expectations of the traffic. While those bits can be leveraged for QoS, a better solution is to use two of the unused bits. These would be ignored by configurations that do not support QoS. If NVMe is used to carry the SCSI command directly, then these bits can be mapped to the "group number" field in SCSI header, which remains unused [26]. In either case, the network mapping protocol (i.e., NVMe-oF) needs to extract these bits and provide it to the QTCP/QRDMA layers on the transmit side, which can use it for appropriate endpoint queuing. This does not suffice for in-network QoS treatment of PM traffic mentioned above in Section 3.1.3. Switches cannot do **deep packet inspection (DPI)** to find the QoS bits from NVMe/SCSI commands; instead, a direct network-level QoS bits are also needed. Since these must be placed in each network packet, it is most appropriate to make use of IP-level QoS bits. In IPv4, these are the ToS (Type of service) or DSCP bits. In IPv6, these are the traffic class bits. The transmit side QTCP/QRDMA will set these in the packet headers following the segmentation of the access PDU into packets. Strictly speaking, only one ToS bit is needed to differentiate between storage and PM traffic in the network, but one could carry the two-bit classification as well. The packet-level bits have no relevance beyond the QTCP/QRDMA receive endpoint, since the NVMe and device-level actions can use the QoS bits in the commands directly.

With priority treatment, there is the issue of how the priorities are assigned by the host. There are many ways to do this, but this aspect is beyond the scope of this article. Generally, the treatment will be derived from the application type in terms of transfer sizes, average intensity, SLA, and so on.

4 PERFORMANCE EVALUATION

In this section, we first propose an E2E QoS differentiating simulator, followed by the evaluation of an NVME-like QoS differentiation inside the device. Finally, we dive into the extensive E2E evaluation of our proposed mechanisms.

4.1 DiffERSim—A QoS Differentiating E2E Remote Simulator

Evaluation of an E2E scenario, which involves the host end, request and response path, storage access protocol, and the device end, is not an easy feat due to the expensive nature of creating such an environment. The academic and open source community relies heavily on detailed simulators, however, existing simulators such as NS3 [35], Omnet++ [41], and MQSim [39] simulate only the network or the storage end (both device and access protocol) and do not take into consideration E2E QoS Differentiation. To evaluate the effect of our proposed methodologies on the E2E behavior of remote applications, we propose a novel QoS **Diff**erentiating **E**2E **R**emote Storage and Memory Simulator—**DiffERSim**. DiffERSim simulates QoS-aware E2E remote storage and memory access. It utilizes a combination of the popular network simulator NS3 and the widely used NVMe SSD simulator—MQSim to create a datacenter environment consisting of multiple hosts connected to multiple storage servers via switches in the network. These storage servers

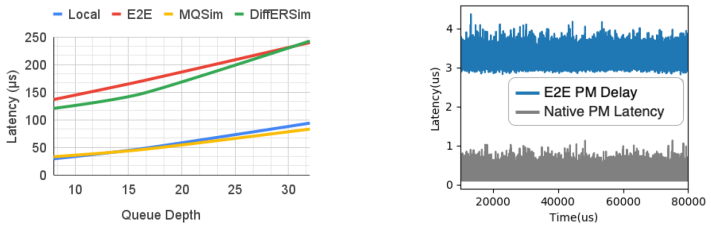


Fig. 5. Evaluation of (a) simulation vs. real environment and (b) PM access in DiffERSim.

themselves contain multiple SSDs and Persistent Memory devices. We extended MQSim's comprehensive DRAM buffer module to simulate a persistent memory device. We embarked on an extensive exercise to ensure DiffERSim supports a wide array of functionalities. Some of these features are:

(1) Simulates both request and response path by utilizing NS3's detailed implementation of the network, i.e., network link, which carries host requests and server responses via switches.
(2) Supports multiple SSDs and multiple PMs with varying device configurations.
(3) Simulates NVMe-oF with multiple transport protocol options—e.g., DCTCP, QTCP, etc.
(4) Contains an inbuilt traffic generator with storage block trace reading functionalities.
(5) QoS differentiation extended to match NVME Weighted Round Robin queue arbitration principle. Thus, ensuring QoS awareness at the storage access protocol level.
(6) Plug and play address translation module to map requests to the appropriate device.

To evaluate DiffERSim's performance in comparison to a real E2E scenario, we compared real-life storage accesses (both remotely and locally) with their simulation equivalent configurations in Figure 5(a). We used a storage workload to request data from a "Samsung 970 EVO Plus" SSD, both locally and over the network (with a 100 Gbps link) using TCP as its transport protocol. We also evaluated the same experimental setup in our simulation environment. From Figure 5(a), we can observe that the MQSim SSD simulator (which DiffERSim utilizes as a module to simulate a single SSD) performs almost identically as accessing a single SSD locally. Both real and simulated SSD access in this case report latency values in the range of 40–90 μs. In Figure 5(a), we also see that DiffERSim's E2E performance compares to the latency values exhibited in a real E2E storage access. This shows that DiffERSim simulates both local and E2E storage access latency accurately.

In Figure 5(b), we also evaluated the PM access performance in DiffERSim by using its inbuilt traffic generator to request memory traffic over the network. In the case of accessing PM locally as a memory device, it has been reported [11] that its access latency ranges in hundreds of nanoseconds. In DiffERSim, we observed (as shown in Figure 5(b)) that the native PM latency at the endpoint averages <900 ns, while the E2E delay for the remote memory traffic falls in the range of 3–4 μs. This is an accurate representation of the performance of PM traffic, both remotely and locally as the latency values quoted in the *SNIA PM Summit* [15] talk about how accessing Persistent Memory over the Fabric results in latency values <4 μs.

For the subsequent evaluations of our proposed strict provisioning for PM traffic and QTCP, QRDMA, we utilize the detailed open sourced NS3 modules of DCTCP (which closely follows RFC 8257) and DCQCN. For the datacenter topology, we use the fat-tree topology as shown in Figure 6. It is to be noted that in spite of the existence of numerous topologies, data centers still widely use the fat-tree topology. As depicted in Figure 6, there are three levels of switches—edge, aggregation, and core switches, i.e., the maximum number of hops for each request or response is 6. We use

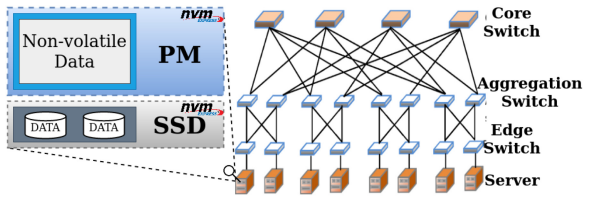


Fig. 6. Fat-tree architecture with simulated storage servers in DiffERSim.

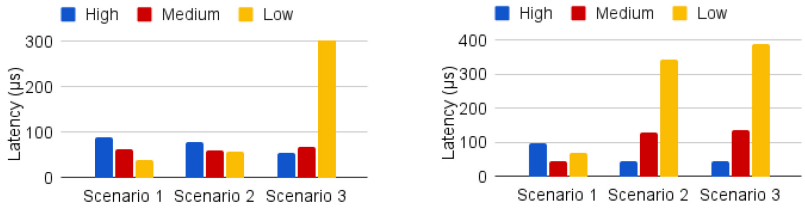
100 Gbps links for all experiments and utilize a mixture of read and write traffic using DiffERSim's inbuilt traffic generator. We evaluate QoS configuration for storage traffic, both in isolation and with PM traffic, to observe the effect of strict provisioning for memory traffic on storage traffic. However, we first explore the need for in-device QoS differentiation at the storage end.

4.2 Evaluation of QoS Differentiation at the Storage End

In this section, we show how our proposed changes in Section 3.2 could be used to provide latency differentiation in the storage end. We consider three cases based on the treatment each QoS class receives:

- **Scenario 1:** QoS Agnostic Protocol and Device, i.e., both NVMe and the SSD treat all flows equally without any QoS differentiation.
- **Scenario 2:** QoS Aware Protocol and QoS Agnostic Device, i.e., NVMe provides WRR differentiation though the SSD queues (i.e., device-level queues) do not respect these priorities.
- **Scenario 3:** QoS Aware Protocol and Device, i.e., device queues respect NVMe priorities too.

We ran the storage simulation component of DiffERSim in isolation to observe the behavior of storage workloads in the three different scenarios in Figure 7. We utilize the workloads published in the SNIA repository [33] by Fujitsu Labs for the three classes of applications. The Fujitsu Lab traces are read-intensive VDI workloads consisting of wide variations of request sizes. The High-, Medium-, and Low-class applications are in decreasing order of intensities. This results in the average latency of the High QoS class exceeding 85 μs for *Scenario 1* in Figure 7(a). The Medium and Low classes performed significantly better, reporting latencies less than 70 and 40 μs, respectively. This is because no differentiated treatment is provided to the workloads. There is no prioritization and the high intensity of the High QoS class results in it performing significantly worse. Following this, we engaged the WRR queue arbitration in NVMe for *Scenario 2. It is to be noted that for all our evaluations, the priority weight between the three classes is proportioned as 3:2:1.* In *Scenario 2*, we observed that the latencies for both the High and Medium classes reduced while the Low-class traffic performed worse than it did in *Scenario 1*. The gap in performance between the three QoS classes reduced, but as we can see in Figure 7(a) *Scenario 2*, it is not enough as the High QoS class still performs worse than the other classes. This is since the queuing latency on the interfacing side (i.e., NVMe submission queues) is not the only significant component contributing to the storage latency. As mentioned in Section 2.3, queuing inside the SSDs coupled with background activities form a major component of the SSD latency. Hence, we extended the notion of QoS differentiation at the NVMe side to the device-level queues as discussed earlier in Figure 3.2. This simply ensures that the device-level queues also follow the WRR queue arbitration while scheduling transactions to the SSD back end. This simple change brought forth a significant difference in performance for all three QoS classes as we can see in *Scenario 3* for Figure 7(a). We finally observe the desired differentiation between the three classes. The Low QoS class performs significantly worse to ensure that the other two classes perform better with the High QoS reporting an average latency value close to 50 μs, while the Medium class reports close to 60 μs. It is



(a) Fujitsu Lab Trace (b) TPCC Trace

Fig. 7. Average latency comparison for different storage QoS scenarios.

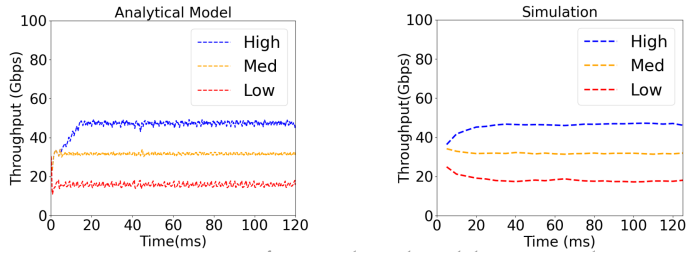


Fig. 8. Comparison of our analytical modeling vs. simulation.

to be noted that we do not consider the storage throughput as for remote storage access, the network throughput is the bottleneck as discussed in Section 1.1. We also see the effect of network throughput on the storage throughput in later sections. We also tested a less pathological case in Figure 7(b) with the three applications being a combination of the TPCC benchmark collected at Microsoft [21]. The intensity of the High- and Medium-class traffic, in this case, is lesser in comparison to an intense Low-class application. We observed that in this case, NVMe-level QoS differentiation is enough to guarantee QoS as seen in *Scenario 2* of Figure 7(c). Extending the QoS notion to the device-level queue does not affect the differentiation provided by the NVMe protocol as shown in *Scenario 3* of Figure 7(c). *Hence, we show that even though the exact latency ratio of 3:2:1 is not achieved, having QoS-aware device-level queues (along with NVMe WRR) is enough to provide storage end-latency differentiation across a wide range of workloads, irrespective of their intensities.* It is to be noted that we do not consider the "urgent" priority for the PM inside the device. This is because the urgent traffic entering the PM device does not need to compete with other classes of traffic, unlike storage traffic in the SSDs. We now look into the performance of our proposed QTCP and QRDMA.

4.3 Evaluation of QTCP and QRDMA

4.3.1 Analytical Model Evaluation. We first compare our analytical model (described in Section 3.1.2) with DiffERSim's implementation of QTCP in Figure 8 using the same topology as Figure 6. We measure the throughput experienced by High, Medium and Low QoS applications, with an offered load of 90, 60, and 30 Gbps, respectively. The bottleneck link has the standard data center 100 Gps link bandwidth. Assuming the threshold value K to be 140 (K being approximately $0.17Cd$, where C is the bottleneck capacity and d is the propagation delay) as mentioned in Reference [3], we observe the results obtained in Figure 8. We see that both the DiffERSim implementation of QTCP and our analytical model exhibit similar behavior, i.e., the carried throughput for all three QoS classes are almost identical. In addition to that the target ratios are also respected in between all the applications. This confirms that our simulation reflects the results exhibited from

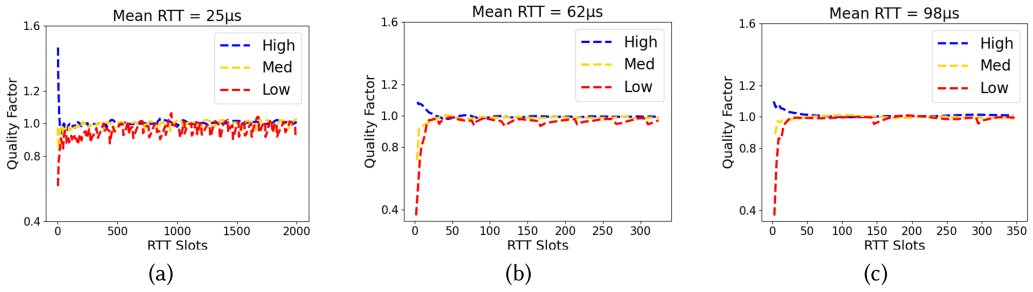


Fig. 9. QF convergence for differing RTTs.

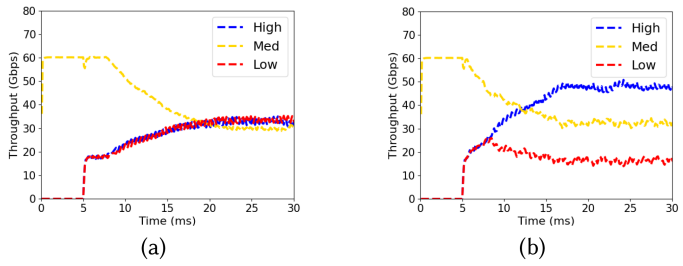


Fig. 10. Convergence comparison between (a) DCTCP and (b) QTCP.

our analytical modeling and hence corroborates the accuracy of DiffERSim's implementation of our proposed mechanism.

4.3.2 Convergence and Stability Analysis. In this section, we evaluate the behavior of our proposed Quality Factor metric (Q_i for a flow i). We utilize our simulation implementation for all our evaluations henceforth. In Figure 9, we observe how stably Q_i behaves for succeeding RTT slots and the time taken for it to converge to unity, i.e., time taken for each QoS class to receive its target throughput. We also observe the same for differing mean RTT values with our simulation setup along with the applications kept the same as Section 4.3.1's simulation setup. In Figure 9(a), we observed that for a mean RTT of 25 μs, it took close to 50 RTT intervals (or 1.2 ms) for Q_i to converge. Similarly for RTT values of 62 and 98 μs in Figures 9(b) and 9(c), it converges within 30–40 RTT slots. The large RTT values result in a slightly larger albeit negligible overall convergence time. Hence, our proposed QTCP differentiation would be able to react and provide differentiation within 1–3 ms. It is to be noted that not all applications take this long to converge—for example, in Figure 9(a), the High-class application converges the fastest (in around 15–20 RTT slots), while the Low-class applications take the longest (i.e., 50 RTT slots). Also, the E2E latency of a request ranges from 3 to 15 ms for High and Low classes, respectively. This shows that the convergence time is just a fraction of the E2E latency of a single request, irrespective of the QoS class. We also observe that after converging, Q_i remained fairly stable within 0.95 to 1 for each class.

In a datacenter environment it is not realistic to assume that all flows start at the same time. Hence, in Figure 10, we also considered a scenario where multiple flows enter the datacenter and cause congestion at a later time. In this case, we plotted the time taken for both DCTCP and QTCP to converge to their appropriate proportioned throughput division with a mean RTT of 98 μs.

We noticed that QTCP converged faster than DCTCP in Figure 10(a) by ~5 ms, i.e., by more than 50 RTT slots. This shows that QTCP provides QoS differentiation faster than DCTCP provides fair treatment. In our previous work [6], we also looked into the behavior of applications with

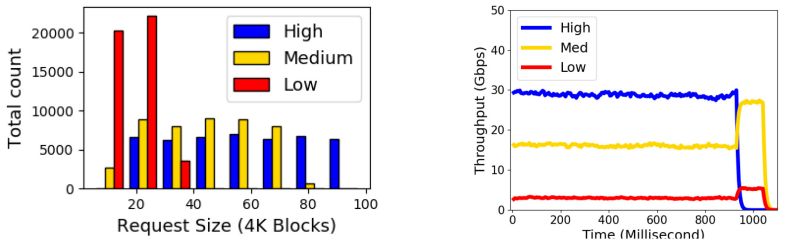
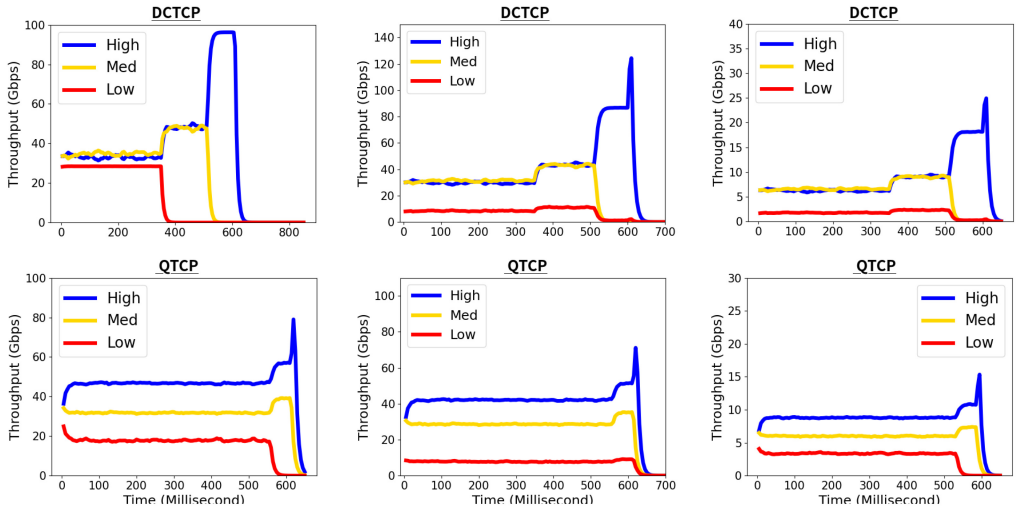


Fig. 11. (a) Workload distribution and (b) evaluation of a scenario with storage bottleneck.

differing RTT values and showed that DCTCP is biased towards flows with shorter RTTs but QTCP converges and stabilizes in spite of the differing RTT values. DCTCP's slow convergence is a widely discussed problem in the literature [4]; we have also observed this in our simulation.

4.3.3 Evaluation of Throughput-sensitive Applications. We now look into the behavior and effect of QTCP in an E2E context. We utilize DiffERSim to create the simulation setup shown in Figure 6 with the network links capacity being 100Gbps. In this testbed the storage end consists of 10 storage servers. Each storage server contains a simulated 1 TB NVMe SSD, which simulates the *Scenario 3* QoS differentiation mentioned in Section 3.2. The applications generate storage traffic following an exponential distribution, aimed towards the storage servers, and send them out using the NVMe-oF protocol. The SSDs in the storage server process the requests and send back the responses to the requesting hosts. In case of writes, the data to be written is sent out on the forward *sender side* link, which the SSDs process, and a response is sent to notify the host that the write request is complete. In the case of reads, the request sent is processed by the SSD and the data is sent back via the *response side* link. In this section, we simulate three workloads of significance—a write-only workload, a read and write mixed workload and a read-only workload. The use of a write-only or read-only workload helps in creating a pathologically overloaded request or response network link.

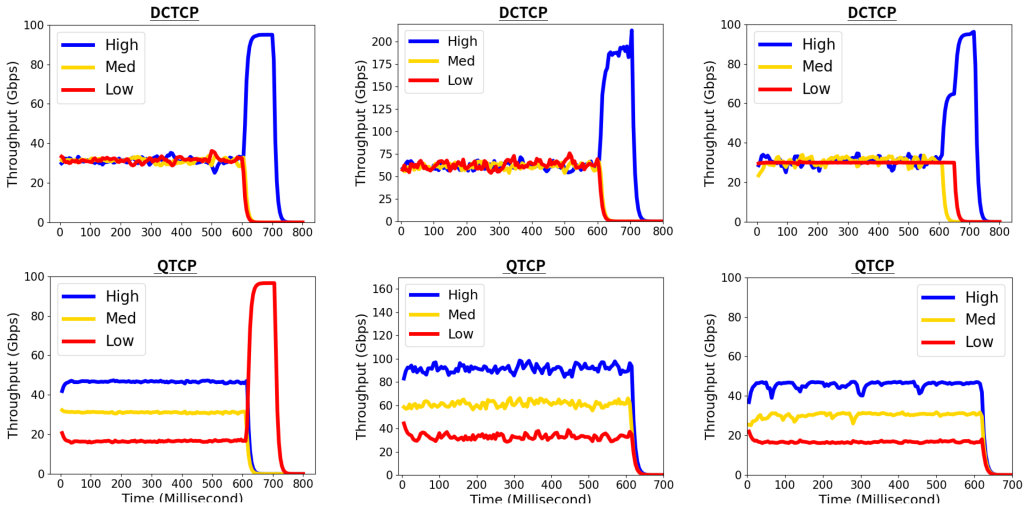
For our throughput-sensitive performance evaluation, the three applications pertaining to High, Medium, and Low QoS classes push 90, 60, and 30 Gbps of write traffic over the 100 Gbps network link. The requests for all classes are in the size of 4K blocks as that is the standard request size for storage workloads. The size distribution for the three classes is shown in Figure 11(a) with the Low-class request size ranging from 5–30 blocks, Medium ranging from 13–75 blocks, and the High class ranging from 17–98 blocks. Both read and write request sizes follow the same distribution and hence only the write size distribution is shown. In Figure 12, we compare the throughput differentiation provided when NVMe-oF uses two different transport protocols, i.e., DCTCP and QTCP. We see the treatment received by the throughput-sensitive write-only storage traffic in an E2E context. For write-only traffic, the bottleneck arises on the sender side. We notice in Figure 12(a) that for DCTCP, all three receive close to identical treatment, i.e., the observed throughput for the three applications is close to 33 Gbps. This is because DCTCP treats all flows the same during congestion. It is to be noted that the Low-class application pushes only 30 Gbps of traffic and hence receives the entirety of its required bandwidth (due to it being less than the DCTCP divided throughput of 33 Gbps). However, our proposed QTCP divides the bottleneck bandwidth into 3:2:1 ratio approximately, thus providing the required QoS differentiation to the three classes. High class receives close to 48 Gbps, with Medium and Low receiving 31 and 17 Gbps, respectively. In Figure 12(b), for DCTCP, we see that despite having storage end differentiation, it is not enough as the network is the bottleneck. It still attempts to provide differentiation but ends up



(a) Sender Side (b) Storage End (c) Response Side

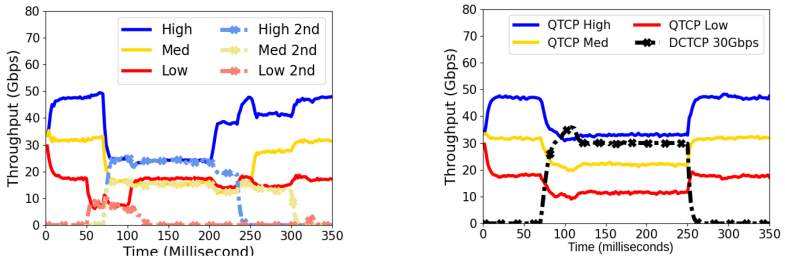
Fig. 12. E2E behavior of a write only workload.

squeezing the Low-class traffic to provide others better treatment. But in the case of QTCP, due to the network-level QoS differentiation, the storage end respects the incoming QoS differentiated traffic. As explained before, the response side in Figure 12(c) just carries the response from the storage end to the hosts to signify that the writes are completed, hence no bottleneck arises on the response side. But again, if the incoming load from the storage end is differentiated itself, then the bottleneck-less link respects the differentiation—as in the case of QTCP. It is to be noted that we simulated the same scenario for DCTCP but with a bottlenecked storage end by reducing the number of storage servers to three. We noticed in Figure 11(b) that in this case even though the forward sender link does not offer QoS differentiation, the storage end still manages to differentiate based on QoS class. However, this scenario is rare as usually, the bottleneck is the network. We now look into applications pushing a mixture of approximately 49.1% reads and 50.9% writes in Figure 13. In this case, the bottleneck is present in both the forward sender side and the backward response side. This is because the applications are pushing 90, 60, and 30 Gbps worth of write data on the forward sender link according to their QoS classes and they are requesting the same amount of data on the backward response link. This results in an extreme congestion episode. In the case of the applications using NVME-oF with DCTCP as the transport protocol, we can see that the High QoS class suffers the most. This is because this application is squeezed the most and hence takes a longer time to complete both the reads (in Figure 13(c)) and writes (in Figure 13(a)). However, in the case of QTCP as the transport protocol, we observe for the sender side in Figure 13(a), all three classes receive the appropriate 3:2:1 QoS differentiation for their offered throughput. The writes for the Low class take slightly longer to complete due to it being squeezed the most. The storage end in Figure 13(b) again respects the QoS differentiated offered load due to no bottleneck. However, unlike Figure 12, in this case, the response side is a bottleneck too due to the existence of reads. Hence, in Figure 13(c) QTCP provides the necessary QoS differentiation to the experienced bandwidth for each application. We also performed the same experiment with a read-only workload, but we have omitted that experiment from this article due to its similarity in behavior with the other experiments (i.e., QTCP succeeds in providing differentiation while DCTCP doles out fair treatment to all classes).



(a) Sender Side (b) Storage End (c) Response Side

Fig. 13. E2E behavior of a read and write intensive workload.



(a) (b)

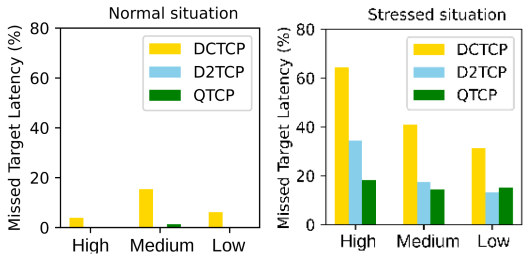
Fig. 14. Evaluating QTCP with (a) multiple intermittent flows and (b) interfering DCTCP traffic.

4.3.4 Adaptive Nature of QTCP. We now investigate QTCP's adaptive nature in case of changes in the datacenter environment. We tested a scenario where applications of different QoS classes start and stop in between a simulation period. Figure 14(a) depicts this scenario, where we have one flow per class present in the datacenter before other flows enter the overall traffic. We introduced another Low-class application around the 50-ms mark into the simulation. This resulted in the bandwidth allocated to the Low class being shared between the two applications pertaining to this class, while the other classes remain unaffected. Similar behavior is observed for the other two classes. This is due to the fact that according to our proposed analytical model in Section 3.1.2, the target throughput is assigned on a per class basis and not a per flow basis. Hence, the introduction of a new flow pertaining to a QoS class only results in the target throughput of that class being shared between all flows of that class. We also see in Figure 14(a) that as soon as an application completes its requests, the other flows pertaining to that QoS class grab on to its assigned target throughput.

We also consider the case where not all competing flows may require QoS differentiated treatment. These non-QoS-sensitive flows could run on DCTCP and may be intermittent too. We make modifications to our proposed QTCP algorithm to accommodate this scenario. QTCP continuously

$W_i = W_i(1 - \alpha/2)$
if $(Q_i - 1.0 > \sigma))$ factor = $-\sigma$;
else if $(Q_i - 1.0 < \sigma)$ factor = σ;
else factor = 0.0
$\lambda = (\lambda + \text{factor}) \times \text{target}_i$
$\text{target}_i = \text{ratio}_i \times \lambda$;
$Q_i = \text{target}_i/\text{actual}_i$;
if $(Q_i < 1)$ $W_i = W_i Q_i$

(a)



(b)

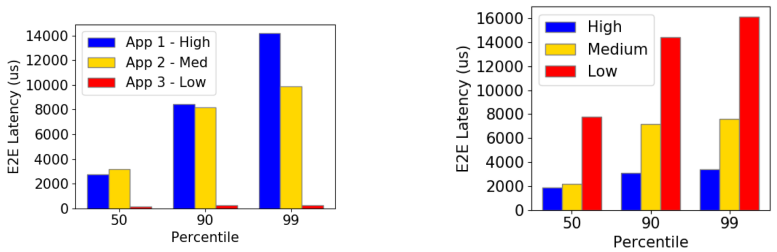
Fig. 15. (a) Algorithm for handling interfering flows and (b) deadline comparison.

measures the interference of the other flows and adjusts the bottleneck bandwidth λ accordingly. We assume that λ is known initially (can be given or even estimated using methods detailed in Reference [7]). If the aforementioned non-QoS-sensitive flow alters the value of λ, then each of the QoS-sensitive flows recognizes this perturbation as detailed in Figure 15(a). The target throughput requirement of a flow i is defined as target_i and the measured throughput is actual_i. Quality Factor converges to its steady state value of 1 pretty fast (as explained in Section 4.3.2) and hence any variation in that value by a value greater than σ (for our experiments the value for σ is taken as 5% or 0.05) is understood to be caused by an interfering flow - either due to its interference with these flows or due to the disappearance of interference. This results in target_i being re-calibrated by an amount "factor" to depict the interference and the window size is hence modified accordingly. In the given algorithm, the fraction of the bottleneck bandwidth assigned is defined as $ratio_i$ and the current measured Quality Factor is depicted as Q_i. The current congestion window for flow i is W_i.

We evaluated the proposed modification in Figure 15(a), by simulating a scenario in Figure 14(b) where an interfering DCTCP flow enters and leaves in the middle of a simulation involving three different applications pertaining to the three QoS classes. We can see in Figure 14(b) that as the DCTCP flow enters the traffic around the 75-ms mark, the QoS flows back off and then again increase their measure flow rates when the interfering flow leaves around the 250-ms mark. One could argue that reserving resources for the DCTCP flow can also be a solution, however, this would result in under-utilization of the network bandwidth when the DCTCP flow is not present.

4.3.5 Evaluation of Latency-sensitive Applications. We now move on from throughput-sensitive applications to look at latency-sensitive applications. We first compare our proposed mechanisms in the network with existing solutions. The QoS classes remain the same as our previous experiments, with their target latency values being defined as 5, 6, and 8 ms, respectively. The mean transfer size of the traffic is 2 MB while the flows follow a Poisson distribution with an average utilization of 80%. We compare the performance of QTCP with DCTCP and D^{2}TCP [40] in Figure 15(b)—the latter is another QoS-based window modulation mechanism, which takes into consideration the "deadline," i.e., the target latency of each class/flow. D^{2}TCP calculates a ratio d_i between the measured delay and the target delay for a flow i and modifies the window size using $\alpha_i^{d_i}$ instead of α_i. We refrain from comparing with other deadline aware mechanisms such as D3 as it has been extensively evaluated to show that D^{2}TCP performs better than D3 in almost all scenarios.

In Figure 15(b), we compare two different situations—normal and stressed situations. The former situation deals with High and Medium applications generating traffic at a frequency lower than the others—10% and 20%, respectively, while in the stressed scenario every class contributes



(a) E2E Latency w Uncoordinated QoS (b) E2E Latency w Coordinated QoS

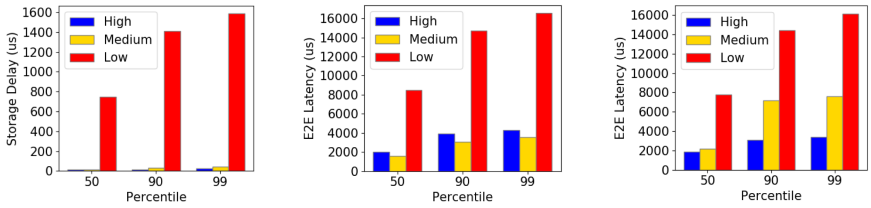
Fig. 16. Effect of coordinated QoS treatment between storage and network end.

equal load. We can see for the normal situation in Figure 15(b) that for almost all the applications, QTCP meets the deadline while DCTCP under performs in comparison, with ∼5–8% packets missing their deadlines. D^{2}TCP meets all the deadlines in the normal situation but it is to be noted that as the higher class of applications increase their frequency of traffic generation in the stressed situation, D^{2}TCP starts falling behind QTCP, with QTCP reducing the percentage of missed deadlines from ∼35% to ∼18% in the case of the high class application. For the stressed situation, QTCP in comparison to DCTCP reduces the fraction of traffic with missed target latency from ∼30–65% to ∼15–18% (i.e., ∼71% reduction as compared to DCTCP).

We now look at the tail-latency performance of QTCP for an E2E context. It is to be noted that congestion in the case of latency-sensitive applications is assumed to be transient and fleeting in nature. This is due to the fact that in the case of longstanding congestion, E2E latency of requests can end up being ever increasing because of the long wait times in send/receive queues. Hence, in all of our subsequent E2E evaluations of latency-sensitive traffic, we consider simulations lasting 1–5 s, which depicts the time period of a single congestion episode.

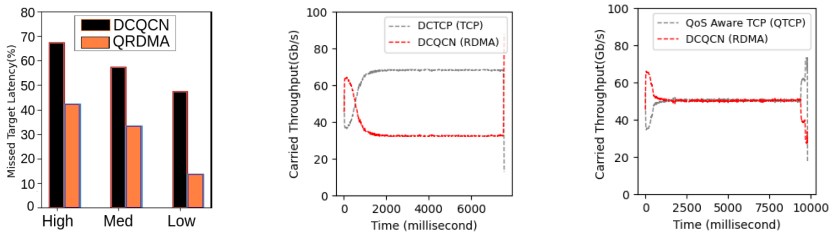
Keeping our DiffERSim setup the same as in Section 4.3.3, we show only the results for a write only workload with the three classes of applications writing data to the SSD at a same rate of 40 Gbps. This creates a stressed scenario with bottlenecks both in the network (network link is 100 Gbps) and in the storage end. From our discussion in Section 2.3, we know that writes are expensive operations in SSD, which in turn can lead to high tail-latency values. We first look into the need for consistent and coordinated E2E QoS treatment in Figure 16. If the storage end is oblivious of the E2E QoS requirement, then it may provide its own notion of QoS at a local level. For example, the FTL in the device may use request size as a metric to determine QoS class of incoming traffic. In Figure 16(a), we assume a scenario where the storage end (i.e., both NVMe and the device) is unaware of the QoS differentiation provided by the network. This also helps us in determining the effect of storage QoS on E2E QoS. In spite of storage latency outperforming the network latency, variability in storage latency (caused by write induced high tail latencies) can affect E2E-latency differentiation. In Figure 16(a), QTCP in the network treats three applications App1, App2, and App3 as High, Medium, and Low classes, respectively. But the storage end assigns its own notion of QoS by treating App1 as Low and App3 as High. This results in App1 and App2 performing worse than App3, despite App 1 requiring the best treatment. We remedy this by ensuring both network and storage provide consistent coordinated QoS treatment in Figure 16(b), where we observe that the desired differentiation is respected.

In Figure 17, we observe the aforementioned write triggered large tail latencies for the Low-class application, where the 50th to 99th percentile latency value ranges from 0.7 to 1.5 ms. This is extremely poor performance for a device whose median latency values are supposed to be less than 100 μs. However, it is to be noted that the storage-level QoS differentiation mentioned in Section 3.2 ensures that while the Low QoS class suffers, High and Medium report values close



(a) Storage Delay (b) E2E Latency w DCTCP (c) E2E Latency w QTCP

Fig. 17. Tail-latency comparison.



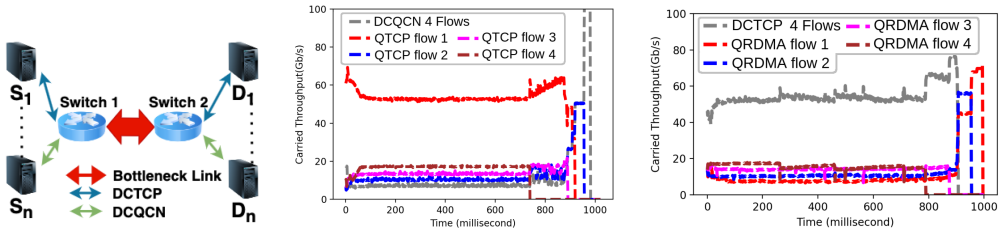
(a) DCQCNvsQRDMA (b) DCTCPvsDCQCN (c) QTCPvsDCQCN

Fig. 18. Latency comparison and bandwidth sharing comparison.

to 60 and 100 μs, respectively. Although the storage end offers QoS differentiation, this is not enough to guarantee E2E-latency differentiation for the different classes. We see that NVME-oF with DCTCP as a transport protocol in Figure 17(b), fails to provide adequate differentiation while QTCP helps in providing adequate differentiation between the three classes in Figure 17(c) (which is a repetition of our experiment in Figure 16(b)). The bottleneck at both the network and storage end (along with this being an E2E evaluation) result in us choosing higher target values. We notice that High class performs better than the Medium class by close to 50%, while the Medium class in turn performs better than the Low class by close to the same amount. Thus, ensuring that relative QoS differentiation is provided. Differentiation is observed across the board of tail-latency percentile values. However, with DCTCP, the High class performs worse than the Medium by close to 20% in all cases. The storage end treats the Low class significantly poorly due to the stressed scenario and hence it performs the worst even in an E2E context with DCTCP.

4.3.6 Evaluation of RDMA Traffic. As discussed in Sections 2.4 and 3.1.1, the feedback-based flow control in DCQCN works nearly identically to DCTCP. Hence, the fundamental workings of QRDMA (i.e., QoS-aware DCQCN) and QTCP remain the same. This behavior was confirmed in our previous work [6], where we observed that for throughput-sensitive applications, DCQCN treats all flows equally during congestion while QRDMA differentiates based on their offered load ratios. Hence, we focus on evaluating latency-sensitive storage flows utilizing DCQCN or QRDMA as its transport.

In Figure 18(a), we evaluated latency-sensitive applications for QRDMA in comparison to DCQCN. The evaluation considers a stressed scenario similar to Figure 15(b). It is observed in Figure 18(a) that QRDMA outperforms DCQCN, with QRDMA having ~43–80% fewer deadline misses in comparison to DCQCN. However, QRDMA's latency deadline misses are more than QTCP's in Figure 15(b). This is because QRDMA is based on DCQCN and DCQCN requires parameter tuning. We have used the default parameter settings noted in Reference [48] as parameter tuning is out of the scope of this article.



(a) (b) (c)

Fig. 19. (a) Evaluation setup for coexisting flows and (b, c) comparison between coexisting flows.

Further, from our discussion in Section 3.1.3, we know that remote PM traffic utilizes QRDMA as a transport. It is unrealistic to assume that said PM traffic exists in isolation in a datacenter environment. Due to this need for coexisting DCTCP and DCQCN traffic (along with QRDMA's observed similarity in performance with QTCP), we refrain from carrying out QRDMA's E2E evaluation and instead focus on the coexistence scenario. We first discuss the need for our Quality Factor metric in such a mixed traffic scenario.

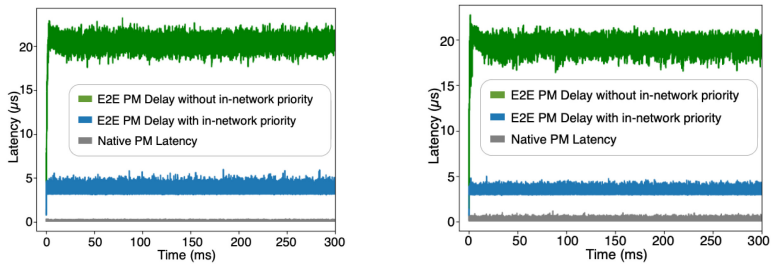
We first look into the bandwidth sharing between DCQCN and DCTCP followed by QTCP and DCQCN in Figures 18(b) and 18(c). We utilize the topology shown in Figure 19(a) where a bottlenecked 100 Gb/s link is shared by DCTCP and DCQCN traffic. Figure 18(b) shows the bandwidth distribution when DCTCP and DCQCN are used with optimal parameter settings. Initially, DCQCN grabs most of the link bandwidth while DCTCP goes through the slow start phase, but eventually it suffers. This results from the somewhat different evolution of DCTCP and DCQCN windows.

In the case of DCTCP, no ECN in the last window causes the congestion window size to increase by 1. In terms of rate increase, if we define it as R_{AI}, then during the congestion recovery phase, current rate RC_i for DCTCP would be $RC_i = RC_i + R_{AI}$. However, DCQCN follows a fast recovery phase, which essentially increases its current rate RC_i by approximately $RC_i + \frac{R_{AI}}{2}$, which is less than the DCTCP case. Hence, initially DCQCN flows grab much of the bandwidth but during fast recovery, DCTCP rate increase is higher than the DCQCN. Thus, DCTCP flows gradually end up grabbing most of the bandwidth and stabilizes. Our experiment in Figure 18(b) corroborates this analysis, which is due to DCQCN's optimal parameter settings that keeps queue depth below the threshold. The QoS-based adjustment of transmission rates using the Quality Factor metric fixes this problem as shown in Figure 18(c) and both get equal bandwidth under congestion. Note that original DCQCN suggests bandwidth reservation for the TCP traffic, to provide fairness for DCQCN flows [48]. But that may lead to under-utilization during non-congestion period.

Notice that the notion of quality factor can also be used to differentiate QoS for individual flows. Using the topology in Figure 19(a) for our evaluation, we set the relative ratio of flows as 1:1.5:2:3. We observe the QoS differentiation within the QTCP flows while also making way for the remaining DCQCN flows in Figure 19(b). We can see similar behavior for DCTCP with QRDMA flows offering the same relative ratio of offered load in Figure 19(c). The ECN parameter setup is in favor of the optimal performance of DCQCN, not QTCP, thus this makes respecting the target throughput difficult in Figure 19(b). However, in the case of standalone DCTCP flows with QRDMA flows in Figure 19(c), the ratio is approximately equal to the target throughput ratio.

4.4 Evaluation of Mixed PM and Storage Traffic

It is to be noted that for practical reasons, only one of QTCP or QRDMA is likely to be configured or enabled in the datacenter unless there is a real need to have both. Mixed PM (which is best served by QRDMA) and storage traffic (which is then better carried by QTCP) is one such scenario. A mixed



(a) Read Only Workload (b) Write Only Workload

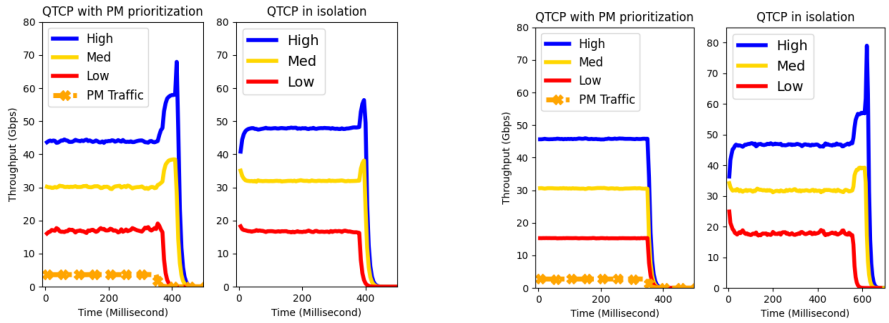
Fig. 20. Comparison of E2E latency for PM traffic.

QTCP and QRDMA storage traffic scenario may also arise but that is beyond the scope of this work. In this section, we evaluate the behavior of small-sized PM traffic (using QRDMA) sharing the same bottleneck link as the larger-sized storage traffic (using QTCP). For this experiment, the PM traffic is Poisson with a size of 128B or 256B with equal probability while the storage access size follows the same distribution as in Figure 11(a). Also, **Persistent Memory over Fabrics (PMoF)** [15] using RDMA is another solution to efficiently support remote persistent memory. However, in this work, we have considered NVMe-oF (using QRDMA) for our remote PM evaluation as both NVMe-oF and PMoF are almost identical from a simulation point of view.

We evaluate two different QoS-sensitive mixed traffic scenarios—one with read-only traffic and one with write-only traffic in Figures 20(a) and 20(b), respectively. The accompanying workloads follow the same distribution as in Section 4.3.3. We observe that for both types of workloads, having no in-network strict priority for the PM traffic results in it experiencing E2E-latency values greater than 20μs. This is undesirable due to the low latency requirement of the PM traffic. We can see that the PM access latency itself is in the range of 300–800 ns. Grun et al. [15] talk about how we can attain close to 3ms in E2E latency for PM traffic by using RDMA. Figure 20 shows similar performance with QRDMA as the transport. We see that our in-network prioritization (proposed in Section 3.1.3) results in 82% improvement, with the E2E latency ranging from 3–4 μs consistently. This is despite the presence of other QTCP traffic, i.e., High-, Medium-, and Low-class QoS-sensitive traffic. As mentioned in Section 3.2, even though PM receives "urgent" priority in the network and also at the NVMe level, the device itself does not contain the need to provide QoS differentiation.

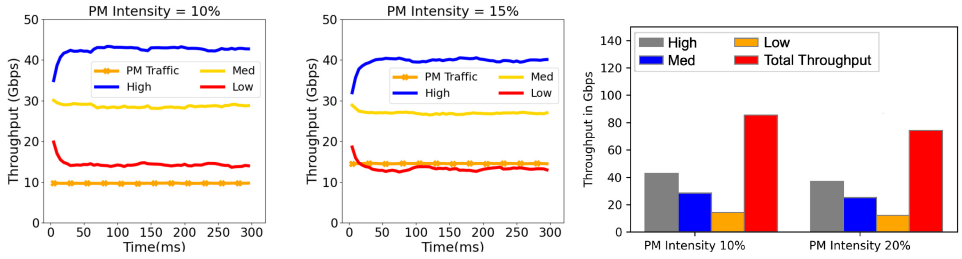
4.4.1 Effect on Throughput-sensitive Flows. In Section 3.1.3, we talk about buffer reservation for PM traffic packets to make way for strict priority small transfer memory traffic. This in turn brings forth the question as to what effect this reservation has on throughput-sensitive large transfer traffic. We observe in Figure 21 the treatment received by the throughput-sensitive QTCP storage flows (High, Medium, and Low classes) in the background for the evaluation shown in Figure 20. The bottleneck link is 100 Gbps while the offered load for the three large transfer applications are 90, 60, and 30 Gbps, i.e., a ratio of 3:2:1. We show only the bottlenecked response path for the read only workload in Figure 21(a) and the bottlenecked request path for the write only workload in Figure 21(b).

In Figure 21, we see that the effect of the PM traffic is negligible due to its small request size. Not having to make way for the small transfer traffic (when QTCP is in isolation) in Figures 21(a) and 21(b) only increases the throughput of each flow by close to 1 Gbps. It is also to be noted that in spite of the presence of the strict priority memory traffic, the throughput-sensitive traffic still receives differentiation in the offered ratio due to the use of QTCP. Similarly, even on increasing



(a) Read Only Workload (b) Write Only Workload

Fig. 21. Effect of PM traffic.

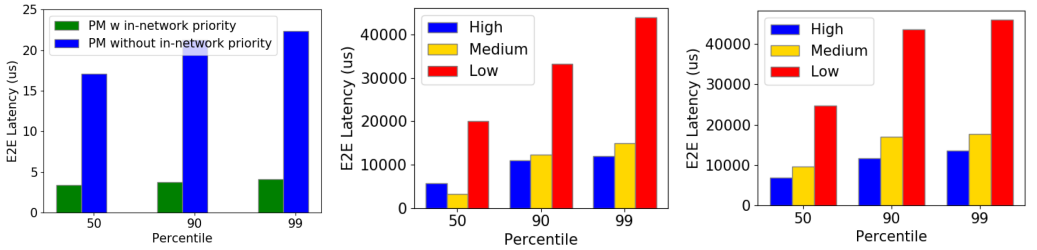


(a) (b)

Fig. 22. Effect of PM traffic with (a) different intensities and (b) incast degree 1:12.

the intensity of the PM traffic in Figure 22(a) to 10% and 15%, the carried throughput reduces slightly but QTCP ensures the QoS differentiation for the other flows is respected. We finally test the bandwidth distribution of throughput-sensitive flows with a different incast degree (1:12) in Figure 22(b), for a PM traffic of 10% intensity. We observe that our proposed methodology still holds its ground with throughput differentiation still followed and the total carried throughput being 87 Gbps (remember, PM takes up 10% of the bottleneck link). Similar observations can be made for a larger intensity of 20% keeping the incast degree same.

4.4.2 Effect on Latency-sensitive Storage Flows. We now look at the E2E behavior of traffic comprising of PM traffic and latency-sensitive storage traffic in Figure 23. We keep the same simulation setup along with the same workloads as discussed in our E2E evaluation in Section 4.3.5. We evaluated the tail latency for all four of the applications in two separate scenarios. Similar to our evaluation in Figure 20, in the first scenario considers the PM traffic is given *no strict priority* while the second scenario is the same but *with strict priority*. In Figure 23(a), we observe similar behavior to what we observed in Figure 20. With in-network priority, the latency observed is fairly consistent within the range of 3–4 μs, across the 50th, 90th, and 99th percentile. However, when there is no in-network priority, the latency increases from 17 to 23 μs. Thus, our proposed change improves PM tail latency by 83%. We also observed that having strict priority for PM traffic slightly increases the observed latency for other latency-sensitive storage flows in Figure 23(c) as compared to Figure 23(b). For larger percentile values, the increase in latency was minimal. For example, the High QoS class latency value changed from 10 ms to close to 11 ms for the 90th percentile. Similarly, for the Medium class it increased from 13 to 15.3 ms. The increase in latency is expected as there is no free lunch, i.e., strict buffer reservation for one flow (or even the existence of another



(a) PM Traffic (b) Storage Flows w/o PM priority (c) Storage Flows w PM priority

Fig. 23. Tail-latency comparison of mixed traffic.

flow) in a congested scenario would always hamper the performance of other flows. But the QoS differentiation is still maintained in spite of the slight increase in latency due to QTCP's stable reaction to background traffic.

5 RELATED WORK

Below, we look into some other works and their limitations pertaining to network and storage QoS.

5.1 Storage QoS

Storage servers in the datacenter often deploy a variety of data management techniques such as sophisticated I/O scheduling mechanisms, tiering/caching across the device hierarchy, and so on. All of these modules provide QoS differentiation opportunities to intelligently manage the access latency; however, QoS differentiation in these contexts is not widely researched or practiced. He et al. talk [18] about a caching methodology that utilizes applications' SLA metrics (e.g., response time) to evaluate caching candidates. It utilizes "re-cache" likelihood by weighting evicted data and adjusting it depending on their performance. Our previous work [17] looked into bridging the gap between succeeding storage layers by caching primarily for the slower device, thus essentially providing High class treatment to requests that may incur a higher average latency. Elnably et al. [14] delve into how storage servers being multi-tiered systems, face challenges in providing QoS to clients. It proposes measuring response times at the host end to determine a reward allocation approach based on References [12, 13]. The proposed algorithm favors clients that are less expensive on the backend storage device by having a static weight assigned to the client and by calculating the elapsed time between the time the request was dispatched and the time at which it is completed. pTrans [32] is another I/O scheduling algorithm that distributes tokens based on QoS reservation requirement, request demand, and storage server capacity, i.e., its availability. It allocates tokens, which is synonymous with the number of requests a client can make depending on the solution of an Integer Linear Programming. However, Kim et al. [22] proposes using the *host memory buffer* as a fast track for processing urgent I/O requests, instead of sensing these urgent requests into the SSDs through a legacy I/O path. Gugnani et al. [16] design a QoS-aware NVMe emulator that provides support for weighted round robin and deficit round robin arbitration mechanisms. However, none of these works delve into the QoS of the requests once they enter the device. Additionally, these works do not discuss network QoS.

5.2 Network QoS

Network QoS is a deeply examined topic, especially in the context of Ethernet-based networks. Since our focus is E2E QoS in datacenter networks, the most relevant related work concerns QoS provisioning for lossless transport protocols. Although the main goal of conventional

transport layer solutions (either TCP or ROCEv2) is fairness (equal sharing of bottleneck bandwidth) amongst different flows, some variants address the differentiated treatment. Differentiated treatment in the Transport layer can be classified into two categories:

Credit-based Control: Expresspass [8] and Reflex [25] are credit-based congestion control mechanisms, which offer differentiated admission control by generating tokens in proportion to the target requirement. The transmitter, receiver, and switches coordinate to control the credit packets (tokens) per-flow basis, which essentially determines the available bandwidth for data packets in the reverse direction. However, credit-based solution requires changes in the protocol and specialized hardware to support token exchange operations. In References [46, 47] the authors have focused on QoS aware flow admission control; however, these studies are not in the NVMe transport context.

Flow Rate-based Control: To the best of our knowledge, no work addresses the QoS issue in data center RDMA transport. But unlike QoS aware RDMA, several works address the issue of QoS aware differentiated flow control in the TCP context. Homa [27], L^2DCT [28], D^2TCP [40], PDQ [19], D^3 [43] consider QoS in terms of individual flow completion time (i.e., deadline). Homa [27] tackles the **head-of-the-line (HoL)** blocking problem that TCP streams present by using in-network queue priority to give small messages (<1,000 bytes) low latency. However, it does not consider the effect of this on other QoS-sensitive traffic. In a real-world scenario, small transfer traffic does not exist in isolation. L^2DCT [28] and D^2TCP [40] adjust the TCP congestion window for various flows based on the stated QoS parameter. One of the main problems with these methods is that to define the QoS parameters, the administrators need to be aware of the network delay and RTT. In contrast, QTCP only needs the relative bandwidth ratio of various flows. PDQ [19] proposes distributed scheduling algorithm, where the switches coordinate among themselves to schedule the high priority flow earlier (i.e., the flow with a critical deadline). However, PDQ needs specialized switches and protocol header additions to transmit the QoS indications. Another deadline aware TCP variation is D^3; however, D^3 [44] needs specific switches, making it impractical as a general solution. Since D^3 also needs centralized control, the communication overhead could have a negative impact on scaling.

6 CONCLUSIONS

In this article, we discuss the end-to-end QoS differentiation for networked storage systems by coordinating together our proposed network transport QoS solution along with existing NVMe-level differentiation and NVMe-like differentiation in the storage device. We also consider the coexistence of remote persistent memory (PM) and storage traffic in the network and show how the PM traffic can achieve the very low latency that it required without the need for any specialized hardware. We demonstrate that a consistent configuration of QoS at all resources in the networked storage path is essential to achieve expected differentiation and performance. In this work, we have considered only a single class for the PM traffic that uses a low-latency RDMA-based transport, where multiple QoS classes are defined for the storage traffic that uses TCP as the transport. In the future, we will consider multiple QoS classes of RDMA-based traffic as well.

ACKNOWLEDGMENTS

The authors acknowledge the support and technical inputs from Dave Minturn for conducting this research.

REFERENCES

[1] A. Alazzaw, A. Pal, and K. Kant. 2020. Efficient big-data access: Taxonomy and a comprehensive survey. *IEEE Trans. Big Data* 8, 2 (2020), 356–376. https://doi.org/10.1109/TBDATA.2020.3036813

[2] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP). *SIGCOMM Comput. Commun. Rev.* 40, 4 (Aug. 2010), 63–74. https://doi.org/10.1145/1851275.1851192

[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM'10)*. ACM, New York, NY, 63–74. https://doi.org/10.1145/1851182.1851192

[4] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. 2011. Analysis of DCTCP: Stability, convergence, and fairness. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'11)*. ACM, New York, NY, 73–84. https://doi.org/10.1145/1993744.1993753

[5] Stephen Bensley, Dave Thaler, Praveen Balasubramanian, Lars Eggert, and Glenn Judd. 2017. Data Center TCP (DCTCP): TCP Congestion Control for Data Centers. RFC 8257. Retrieved from https://www.rfc-editor.org/info/rfc8257. https://doi.org/10.17487/RFC8257

[6] Joyanta Biswas, Jit Gupta, Krishna Kant, Amitangshu Pal, and Dave Minturn. 2021. Provisioning differentiated QoS for NVMe over fabrics. In *Proceedings of the IEEE 46th Conference on Local Computer Networks (LCN'21)*. IEEE LCN, 154–161. https://doi.org/10.1109/LCN52139.2021.9524967

[7] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-based congestion control. *ACM Queue* 14, 5 (2016), 20–53.

[8] Inho Cho, Keon Jang, and Dongsu Han. 2017. Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'17)*. ACM, New York, NY, 239–252. https://doi.org/10.1145/3098822.3098840

[9] D. Cobb and A. Huffman. 2012. Nvm Express and the PCI express SSD Revolution. Intel Developer Forum.

[10] Rob Davis. 2016. The Network is the New Storage Bottleneck. RetrievedfromhttpS://www.datanami.com/2016/11/10/network-new-storage-bottleneck/.

[11] Kevin Deierling. 2016. Convergence of Storage and Memory. https://www.snia.org/sites/default/files/NVM/2016/presentations/NVMSummitMellanox_Final.pdf

[12] Ahmed Elnably, Kai Du, and Peter Varman. 2012. Reward scheduling for QoS in cloud applications. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'12)*. IEEE/ACM CCGRID, 98–106. https://doi.org/10.1109/CCGrid.2012.120

[13] Ahmed Elnably and Peter Varman. 2012. Brief announcement: Application-sensitive QoS scheduling in storage servers. In *Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'12)*. ACM, New York, NY, 185–187. https://doi.org/10.1145/2312005.2312040

[14] Ahmed Elnably, Hui Wang, Ajay Gulati, and Peter Varman. 2012. Efficient QoS for multi-tiered storage systems. In *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'12)*. USENIX Association, USA, 6.

[15] Paul Grun, Stephen Bates, and Rob Davis. 2018. Persistent Memory Over Fabrics. https://www.snia.org/sites/default/files/PM-Summit/2018/presentations/05_PM_Summit_Grun_PM_%20Final_Post_CORRECTED.pdf

[16] Shashank Gugnani, Xiaoyi Lu, and Dhabaleswar K. Panda. 2018. Analyzing, modeling, and provisioning QoS for NVMe SSDs. In *Proceedings of the IEEE/ACM International Conference on Utility and Cloud Computing (UCC'18)*, Alan Sill and Josef Spillner (Eds.). IEEE UCC, 247–256.

[17] Jit Gupta and K. Kant. 2020. FussyCache: A caching mechanism for emerging storage hierarchies. *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom'20)*. https://doi.org/10.1109/CloudCom49646.2020.00010

[18] Huihong He, Zhiyi Ma, Hongjie Chen, Dan Wu, Huanhuan Liu, and Weizhong Shao. 2014. An SLA-driven cache optimization approach for multi-tenant application on PaaS. In *Proceedings of the IEEE 38th Annual Computer Software and Applications Conference (COMPSAC'14)*. IEEE, 139–148.

[19] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. 2012. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'12)*. ACM, New York, NY, 127–138. https://doi.org/10.1145/2342356.2342389

[20] Krishna Kant. 1992. *Introduction to Computer System Performance Evaluation*. McGraw-Hill, New York, NY.

[21] Swaroop Kavalanekar and Bruce Worthington. 2007. Microsoft Enterprise Traces (SNIA IOTTA Trace 130). Retrieved from http://iotta.snia.org/traces/130

[22] Kyusik Kim, Seongmin Kim, and Taeseok Kim. 2020. FAST I/O: QoS supports for urgent I/Os in NVMe SSDs. In *Proceedings of the 2020 5th International Conference on Intelligent Information Technology (ICIIT'20)*. ACM, New York, NY, 146–151. https://doi.org/10.1145/3385209.3385221

[23] Taejin Kim, Sangwook Shane Hahn, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. 2018. PCStream: Automatic stream allocation using program contexts. In *Proceedings of the 10th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'18)*. USENIX Association, 17.

[24] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. 2019. Fully automatic stream management for multi-streamed SSDs using program contexts. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, 295–308.

[25] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote flash ≈ local flash. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. ACM, New York, NY, 345–359. https://doi.org/10.1145/3037697.3037732

[26] Michael Mesnier, Feng Chen, Tian Luo, and Jason B Akers. 2011. Differentiated storage services. In *Proceedings of the ACM Symposium on Operating Systems Principles*. 57–70.

[27] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'18)*. ACM, New York, NY, 221–235. https://doi.org/10.1145/3230543.3230564

[28] A. Munir, I. A. Qazi, Z. A. Uzmi, A. Mushtaq, S. N. Ismail, M. S. Iqbal, and B. Khan. 2013. Minimizing flow completion times in data centers. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM'13)*. IEEE, 2157–2165.

[29] NVMe 2019. NVM Express Base Specification, Rev 1.4. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf

[30] NVMeOF 2019. NVM Express over Fabrics Revision 1.1. Retrieved from https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1-2019.10.22-Ratified.pdf

[31] OpenSSD 2022. The OpenSSD Project. Retrieved from http://www.openssd-project.org/

[32] Yuhan Peng and Peter Varman. 2020. PTrans: A scalable algorithm for reservation guarantees in distributed systems. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'20)*. ACM, New York, NY, 441–452. https://doi.org/10.1145/3350755.3400273

[33] SNIA Iotta Trace Repository. 07.15.2020. Systor'17 Fujitsu Laboratory Traces. Retrieved from http://iotta.snia.org/traces/4964

[34] Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh Jagadeesh Shetty, Joo-Young Hwang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2018. FStream: Managing flash streams in the file system. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. USENIX Association, 257–263.

[35] George F. Riley and Thomas R. Henderson. 2010. The ns-3 network simulator. In *Modeling and Tools for Network Simulation*. Springer, 15–34. Retrieved from http://dblp.uni-trier.de/db/books/collections/Wehrle2010.html#RileyH10

[36] Tanaya Roy, Jit Gupta, Krishna Kant, Amitangshu Pal, and Dave Minturn. 2021. PLMLight: Emulating predictable latency mode in regular SSDs. In *Proceedings of the IEEE NCA Conference*. https://doi.org/10.1109/NCA53618.2021.9685772

[37] T. Roy, J. Gupta, K. Kant, A. Pal, D. Minturn, and A. Tavvakol. 2021. Managing SSD tail latency with PLM. In *Proceedings of the NAS Conference*. https://doi.org/10.1109/NAS51552.2021.9605470

[38] Billy Tallis. 2019. The Samsung 970 EVO Plus (250GB, 1TB) NVMe SSD Review. Retrieved from https://www.anandtech.com/show/13761/the-samsung-970-evo-plus-ssd-review/5

[39] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. 2018. 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, 49–65.

[40] Balajee Vamanan, Jahangir Hasan, and T. N. Vijaykumar. 2012. Deadline-aware datacenter Tcp (D2TCP). In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'12)*. ACM, New York, NY, 115–126. https://doi.org/10.1145/2342356.2342388

[41] András Varga and Rudolf Hornig. 2008. An overview of the OMNeT++ simulation environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems and Workshops (Simutools'08)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Article 60, 10 pages.

[42] Moshe Voloshin. 2022. *Introduction to Thor Congestion Control for RoCE*. Technical Report.

[43] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better never than late: Meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM'11)*. ACM, New York, NY, 50–61. https://doi.org/10.1145/2018436.2018443

[44] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better never than late: Meeting deadlines in datacenter networks. *SIGCOMM Comput. Commun. Rev.* 41, 4 (Aug. 2011), 50–61. https://doi.org/10.1145/2043164.2018443

[45] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. 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, 207–219.

[46] Timothy Zhu, Daniel S. Berger, and Mor Harchol-Balter. 2016. SNC-Meister: Admitting more tenants with tail latency SLOs. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)*. ACM, New York, NY, 374–387. https://doi.org/10.1145/2987550.2987585

[47] Timothy Zhu, Alexey Tumanov, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. 2014. PriorityMeister: Tail latency QoS for shared networked storage. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'14)*. ACM, New York, NY, 1–14. https://doi.org/10.1145/2670979.2671008

[48] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 523–536. https://doi.org/10.1145/2829988.2787484

Received 15 June 2023; revised 28 October 2023; accepted 29 October 2023