ACHIEVING END-TO-END QUALITY OF SERVICE FOR REMOTE STORAGE AND MEMORY TRAFFIC IN DATACENTERS

A Dissertation
Submitted to
the Temple University Graduate Board

In Partial Fulfillment
of the Requirements for the Degree
DOCTOR OF PHILOSOPHY

---

by
Jit Gupta
December 2023

Examining committee members:

Dr. Krishna Kant, Dissertation Advisory Chair, Department of Computer and Information Sciences
Dr Xubin He, Department of Computer and Information Sciences
Dr Justin Shi, Department of Computer and Information Sciences
Dr. Saroj Biswas, Department of Electrical and Computer Engineering

©
Copyright
2023

by

Jit Gupta

All Rights Reserved

ABSTRACT

Storage servers in the datacenter today comprise low latency and high throughput devices ex. Solid State Drives (SSDs) and Persistent Memory (PM). These can be accessed over the network via storage access protocols such as NVMe-oF (Non-Volatile Memory Express over Fabric) and PM-oF (Persistent Memory over Fabric). Applications accessing these devices may have varying End-to-End Quality of Service (E2E QoS) requirements concerning their carried throughput and latency. Providing differentiated treatment based on these requirements is a challenging task during congestion episodes that may occur at various points in the E2E path, including requests from the host to the storage (via network) and the response traveling back to the host. In this work we explore and propose new techniques that can be utilized to provide E2E QoS differentiation for requests at four major points: (1) the host end from where the requests originate and back to where the responses reach, (2) the request and the response path, (3) the storage access protocol, i.e., NVMe and (4) the storage device.

For the purposes of E2E QoS, we classify requests as either high priority or normal priority. In each category there could be multiple classes that are distinguished by a differentiated ratio of their target throughputs or latencies. For (1), caching in host memory is a technique that can improve the performance of specific QoS classes at the host end. We hence propose a new caching algorithm – FussyCache, that considers the low latency of today's storage devices to make caching decisions.

To achieve differentiation in (2), we propose two new mechanisms called QTCP and QRDMA by modifying the existing congestion control mechanisms in Data Center transport protocols (i.e., DCTCP and DCQCN) to accommodate the notion of QoS. We utilize a ratio between target throughput and measured throughput (or measured latency and target latency) to provide differentiation between the multiple sub-classes within a class, while

we propose reservation-based prioritized treatment for the high priority traffic.

For (3), the NVMe protocol uses the Weighted Round Robin queue arbitration mechanism to provide QoS differentiation to the 4 QoS classes. We utilize this existing technique and extend it to queues inside the device (i.e., the SSD) to tackle non-deterministic access latencies caused due to the internal architecture of SSDs. The NVMe protocol also contains a feature for tackling this issue called the Predictable Latency Mode (PLM). We introduce 2 new mechanisms to tackle the deficiencies of this feature. We first introduce – PLMC, a coordinator module to arbitrate the use of PLM based on QoS classes. Most current SSDs do not have the PLM feature. To address this, we also propose PLMLight, a PLMC-like solution that segregates writes to reduce the possibility of background activity.

Finally, we explore the need for consistent treatment at (1), (2), (3) and (4) based on an application's QoS class, by proposing a request tagging mechanism. We extensively evaluate a combination of our proposed techniques using a variety of workloads to show E2E QoS differentiation. To carry out these evaluations, we built NeSt - a QoS Differentiating E2E Networked Storage Simulator, which simulates a datacenter environment with tens of SSDs along with PMs that can be accessed over the network via switches. We plan to open source this tool to enable further research in E2E QoS for Datacenter Storage and Memory traffic by the research community.

This dissertation is dedicated to my late grandmother, Mrs. Gauri Gupta.

ACKNOWLEDGEMENTS

I firmly believe that one's Ph.D. journey belongs to a large list of people and not a single person. Hence I would like to take this opportunity to acknowledge all who have made this dream possible. I would like to start by thanking my advisor Dr. Krishna Kant for helping me become the researcher I am today. Dr. Kant's demanding nature combined with his knowledge of computer science helped me grow from an individual with no background in computer systems to a person who cannot imagine a life without storage systems and computer networking. I am also grateful to my committee members Dr. Justin Shi and Dr. Xubin He for nurturing my love for Computer Systems. My first two courses taken at Temple University were taught by Dr. Shi and Dr. He. Both of whom were pivotal in helping me understand the details of systems programming and computer architecture. I would also like to thank Dr. Hui Lei and Raja Kommula for helping me gain valuable knowledge and experience during my internships at Futurewei Technologies and Juniper Networks respectively and Dave Minturn for his valuable insights during our collaborations.

I would like to thank my family for showing me the true definition of hard work and determination. My sister, Dr. Ritama Gupta Dempsey has been my biggest cheerleader and has also been my biggest source of inspiration. Her intelligence is second to none and I thank her for being the model sibling that she has always been. I am grateful to my parents Raja and Rajashi Gupta for the sacrifices they have made, the life lessons they have imparted, and for knowing when to be a friend and when to be a parent. Their parenting skills shaped my independent thinking which in turn helped me to ask difficult questions in my research journey. I would also like to thank my friend and brother-in-law Jeffrey Adam Dempsey for being the calm mind I could turn to during stressful times and for helping me settle into the city I call home, Philadelphia.

My PhD life has seen me collaborate with my labmates and other colleagues and I

would like to thank each and every one of them for their contributions in this journey. I would also like to thank Moinak “Jumpy” Ghoshal, a brilliant researcher and my best friend. I have known Jumpy since I was 18 years old and this month marks a decade of our friendship. Conversations with him have ranged from late-night musings about playing guitar solos to debates regarding academia, computer science and politics. He is my go-to person for every obstacle in life and his unwavering support has been a huge source of strength for me. I wish him godspeed on his own PhD journey. I would also like to thank my partner, Dr. Uddipana Kakati. She has made the final year of my PhD life the easiest of them all by being a constant source of inspiration and by showing me what it means to actually enjoy research. Her kindness knows no bounds and there is no one else I can imagine sharing this final lap with.

I would also like to thank my friends Indranil Pradhan, Ushnish Guha Bakshi, Shirsho Dasgupta, Max Milkman, and Darren Chan for keeping me grounded at various points of this journey and for teaching me the importance of not taking life too seriously. I would also like to thank Apurva Ghoshal for helping me navigate the difficulties of surviving a pandemic alone in a different country.

Finally, I would like to end on a slightly unconventional note by acknowledging the four inanimate objects that made every single day of my Ph.D. life worth it - my guitars Sadie, Summer, June and Bubblegum. I will be forever grateful for those countless nights of debugging on GDB while noodling on them. Who would have known that music and computer science are the best combination for a brainwave!

TABLE OF CONTENTS

Page

ABSTRACT iii

ACKNOWLEDGEMENTS vi

LIST OF TABLES xiii

LIST OF FIGURES xiv

CHAPTER

1 INTRODUCTION 1

1.1 Application Quality of Service 3

1.2 Background and Motivation 4

1.2.1 Network Issues in Remote Access 4

1.2.2 High Tail Latencies in Storage Devices 5

1.2.3 Need for QoS in the Host End 6

1.3 Contributions 7

1.3.1 E2E QoS 7

1.3.2 Contributions in Different E2E Path Components 9

1.3.3 E2E Contributions and Outline 10

2 FUSSYCACHE: A CACHING MECHANISM FOR EMERGING STORAGE HIERARCHIES 12

2.1 Overview 12

2.2 Introduction 13

2.3 Related Work 15

2.4 Hit Rate vs. Latency 16

2.5 Implementation Methodology 17

2.5.1 FussyCache Parameters 18

2.5.2 Dynamic Metadata Cache 19
2.5.3 Data Cache 21
2.5.4 Warmup 22
2.6 Implementation of FussyCache 23
2.6.1 Experimental Setup 23
2.6.2 Benchmarks 24
2.7 Results 26
2.8 Conclusions and Discussions 33
2.9 Future Work 34
3 PLMC: TAIL LATENCY CONTROL MECHANISM FOR NVME SSDS IN SHARED ENVIRONMENT 36
3.1 Overview 36
3.2 Background 39
3.2.1 NVMe and Deterministic Latency Feature 40
3.2.2 Predictable Latency Mode (PLM) Operations 41
3.2.3 Deficiencies of latest NVMe PLM 43
3.3 Deterministic Service with Shared NVM-sets 44
3.3.1 Proposed PLM Coordinator 44
3.3.2 Control Architecture of PLM Coordinator 46
3.4 DT Count Allocations 47
3.4.1 Analysis of DC Allocations 48
3.5 Evaluation of the Proposed PLM 50
3.5.1 Enhancing MQSim to Support PLMC 51
3.5.2 Emulation of PLM on Real Server 52
3.5.3 Workloads and Configurations Used 53
3.5.4 Evaluation Results 54

3.6 PLMLight : Emulating Predictable Latency Mode in Real SSDs . . 59

3.7 Evaluation of PLMLight 61

3.8 Conclusions 63

4 CONFIGURING AND COORDINATING END-TO-END QOS FOR EMERGING STORAGE INFRASTRUCTURE 65

4.1 Overview 65

4.2 Introduction 66

4.2.1 Data Center Storage Organization 67

4.2.2 End to End QoS Differentiation 68

4.2.3 Our Contributions 69

4.3 Background and Motivation 71

4.3.1 Host Level Storage Stack 71

4.3.2 High Performance Storage Interfacing 72

4.3.3 Characteristics of Emerging Storage Technologies 74

4.3.4 Network Issues for Remote Storage Access 77

4.4 E2E QoS Differentiation in Networked Storage Transfers 78

4.4.1 QoS in the Network 78

4.4.2 QoS Differentiation at the Storage End 87

4.4.3 End-to-End Configuration of QoS 88

4.5 Performance Evaluation 90

4.5.1 NeSt - A QoS Differentiating E2E Networked Storage and Memory Simulator 90

4.5.2 Evaluation of QoS Differentiation at the Storage End . . 93

4.5.3 Evaluation of QTCP and QRDMA 95

4.5.4 Evaluation of Mixed PM and Storage Traffic 110

4.6 Related Work 114

4.6.1 Storage QoS 115
4.6.2 Network QoS 116
4.7 Conclusions 117
5 NEST : A QOS DIFFERENTIATING END-TO-END NETWORKED STORAGE SIMULATOR 119
5.1 Overview 119
5.2 Introduction 120
5.3 Background and Related Work 122
5.3.1 Storage End QoS 122
5.3.2 Storage vs. Persistent Memory 123
5.3.3 QoS Limitations in Current Data Center Networks 124
5.3.4 Overview of DCTCP and DCQCN 126
5.3.5 Network and Storage Simulators 129
5.4 Proposed Simulator 132
5.4.1 End-to-End Path 132
5.4.2 Multiple Devices with Multiple Configurations 134
5.4.3 SCM Devices 135
5.4.4 QoS Differentiation in the Storage End 135
5.4.5 QoS Differentiation in the Network 136
5.4.6 Integrating Simulators 137
5.5 Comparison with existing simulators 138
5.6 Evaluation 139
5.6.1 Comparison with Real Systems 139
5.6.2 E2E Throughput Evaluation 141
5.6.3 E2E Latency Evaluation 143
5.6.4 Testing NeSt's Trace Replay Functionality 145

5.7 Conclusions 149

6 CONCLUSIONS AND FUTURE WORK 150

6.1 Host End 151

6.1.1 Future Work 151

6.2 Network End 151

6.2.1 Future Work 152

6.3 Storage End 152

6.3.1 Future Work 153

BIBLIOGRAPHY 154

LIST OF TABLES

Table Page

1.1 Difference between SSDs vs SCM Devices 1

1.2 Examples of QoS requirements for Applications 3

2.1 Average Request Size in KB 25

3.1 DTWin attributes involved in PLM 42

3.2 Notation 48

3.3 DataSet Configurations 53

3.4 Parameter Configurations 54

4.1 Common Abbreviations 71

4.2 Description of important terms used in our analysis 84

LIST OF FIGURES

Figure Page

1.1 An End-to-End IO Request Path 2

1.2 QoS Classification 8

2.1 Use of multithreading 23

2.2 Average access latencies (in s) for ENVM 26

2.3 Average access latencies (in s) for L2S2D and Hit Rate for Systor LUN0 . 29

2.4 Comparison of Hit Rates 31

2.5 CPU Utilization 32

3.1 Latency Distribution for Intel DC P4610 SSD 39

3.2 Proposed Coordinator for DTWin Allocation 44

3.3 Tail Latency in isolation vs sharing 53

3.4 Evaluation results for Dataset 1. The tail latency is calculated in microseconds. 55

3.5 Time series for (a)-(b) Utilization and (c)-(d) deficiency characteristics of Dataset1 with Config 2 for strict priority and fixed ratio cases. The 3 QoS classes are represented by Blue: High, Red: Medium, and Grey: Low . . . 56

3.6 Comparison of Median tail latencies for Dataset 1. 57

3.7 Comparison of Median tail latencies for Dataset 4. 58

3.8 An Illustration of PLMLight Architecture 60

3.9 Evaluation of Workload-I 62

3.10 Evaluation of Workload-II 63

4.1 QoS Classification 66

4.2 (a)Overview of end-to-end data center organization and (b) comparison between local and end-to-end storage access 67

4.3 QoS Aware Queuing in NVMe 73

4.4 Internal Architecture of an SSD 74

4.5 Flowchart for Quality Factor (QF) based flow control 79

4.6 Extending NVMe WRR inside the SSD 87

4.7 Evaluation of (a) simulation vs real environment and (b) PM access in NeSt 91

4.8 Fat tree architecture with simulated storage servers in NeSt 92

4.9 Avg. Latency comparison for Different Storage QoS Scenarios with (a) Fujitsu Lab and (b) TPCC traces 93

4.10 Comparison of our analytical modeling vs simulation 95

4.11 QF Convergence for Differing RTTs 96

4.12 Convergence comparison between (a) DCTCP and (b) QTCP 97

4.13 (a) Workload distribution along with (b) evaluation of a scenario with storage bottleneck 98

4.14 E2E Behavior of a Write Only Workload 99

4.15 E2E Behavior of a Read and Write Intensive Workload 101

4.16 Comparison between DCTCP and QTCP for a read only workload 102

4.17 Evaluating QTCP with (a) multiple intermittent flows and (b) interfering DCTCP traffic 103

4.18 Window modulation in presence of interfering flows 104

4.19 Deadline Miss Comparison for DCTCP vs D^2TCP vs QTCP 104

4.20 Tail Latency comparison when QoS treatment is not coordinated 106

4.21 Tail Latency Comparison 107

4.22 Bandwidth sharing comparison between (a) DCQCN vs DCTCP and (b) QTCP vs DCQCN 109

4.23 (a)Evaluation setup for coexisting flows and bandwidth sharing comparison between (b) QTCP w DCQCN flows and (c) QRDMA w DCTCP flows . . 110

4.24 Comparison of E2E Latency for PM traffic for (a) Read Only and (b) Write Only Workload 112

4.25 PM Traffic Effect on Throughput Sensitive Applications for (a) Read Only and (b) Write Only Workload 112

4.26 Effect of PM traffic with (a) different intensities and (b) incast degree 1:12 . 113

4.27 Tail Latency comparison of mixed traffic 114

5.1 Architecture of MQSim 129

5.2 High Level Architecture of NeSt 133

5.3 Comparison of Different Open Source Simulators 138

5.4 Comparison between simulation and real environment 140

5.5 Evaluation of a throughput sensitive read only workload 141

5.6 Evaluation of a throughput sensitive write only workload 142

5.7 Evaluation of a throughput sensitive read write workload 142

5.8 Tail Latency Comparison 143

5.9 Workload (a) read and (b) write distribution 145

5.10 Throughput differentiation of a real workload 146

5.11 Fat Tree Topology 146

5.12 Throughput differentiation in a fat tree topology 147

5.13 Storage Bottleneck Comparison 148

CHAPTER 1

INTRODUCTION

Modern-day organizations cater to millions of users and handle hundreds of petabytes of data pertaining to these users. For example, Netflix reports that it has 230 million subscribers [Ruby (2023)] and that it stores its contents in thousands of servers, each of which contains hundreds of terabytes of streaming content [Syom (2020)]. In order to cater to demands of such a massive scale, organizations tend to handle their data in datacenters that can store, process, and disseminate their data according to the demands of the users and applications pertaining to the organization.

These datacenters consist of sophisticated storage and network equipment to cater to the needs of applications or hosts using it remotely. They contain multiple storage servers in which data storage and memory devices such as Solid State Drives (SSDs) and Storage Class Memory devices (or SCM devices[1]) reside. The difference between them is discussed in Table1.1. The SSDs and the SCM devices (an example of which is Persistent Memory/PM) are accessed via the Non-Volatile Memory Express over Fabric (NVMe-OF) protocol. They are used to store and retrieve data by the hosts over the network remotely via switches and routers (with links of 100Gbps being common). Figure 1.1 shows the

[1] SCM devices are accessible as both storage and memory devices.

Table 1.1: Difference between SSDs vs SCM Devices

Storage Device ex. SSD	**Storage Class Memory Devic ex. PM**
Secondary storage device, using flash memory to store data	Non-volatile byte addressable, low latency memory, similar to DRAM
Sits on PCIe Bus	Sits on DIMM slots
Access latencies in 10-100 s	Provides acccess latencies ¡ 1 s
Can be used as storage only	Can be used as memory or storage
Storage of large volumes of data ex. large databases	Use cases such as fraud detection, IOT and cybersecurity analysis

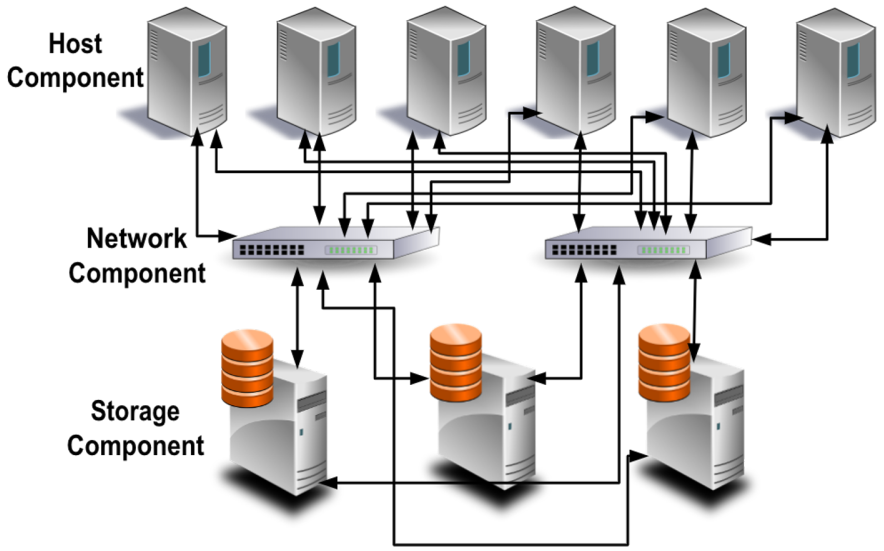

Figure 1.1: An End-to-End IO Request Path

path of a storage request. It is initiated from a host and travels to the assigned *storage volume* via the network, which typically consists of several switches along the path (even though the figure only shows one hop for simplicity). The storage volume is typically a virtual entity and may be mapped across multiple storage devices in the same storage server or across multiple storage servers. Thus the IO requests pertaining to the same storage volume (logical storage volume or LUN) may go to different storage servers as shown in fig. 1.1, however in this work we have mostly considered scenarios where a request maps to a single server.[2] Similarly, a storage server may receive requests from multiple clients. In case of a data-read, the request processing will typically result in a large amount of data transferred back to the host on the reverse path. In case of a data-write, the storage server may receive a large amount of data to be written, followed by an acknowledgment to the host.

Table 1.2: Examples of QoS requirements for Applications

Application Type	Throughput Demand	Latency Tolerance
Email	Low	High
Web Browsing	Low	High
File Transfer (FTP)	Low-High	High
Chat (IM)	Low	Medium
Video Steaming (ex. surveillance)	Medium-High	Medium
Video on demand (ex. YouTube, Netflix)	High	Medium
Voice over IP / WiFi	Low	Low
Video conferencing (ex. Skype, FaceTime)	Medium-High	Low

1.1 Application Quality of Service

As mentioned already, applications that access storage servers in the datacenter are mostly remote. These remote applications have Service Level Agreements (SLAs) which help define their Quality of Service (QoS) requirements [Cisco (2023)] concerning end-to-end throughput and latency[3]. Throughput is the amount of data that can be transferred per unit time while latency is the time required to transfer data. In Table1.2 we see an example of how different applications have varying requirements. For example, an Email application has low requirements with respect to both its throughput and latency while a videoconferencing application like Skype could have high demand for both throughput and latency performance. Hence it is important to consider the entire round-trip path shown in Figure 1.1 for QoS considerations, in order to avoid potential anomalies or sub-optimal QoS treatment.

Additionally, the definition of E2E QoS can encompass a variety of scenarios involving data placement, tiering, application relationships and more. For example, applications using the microservice architecture may have directed acyclic graphs (DAGs) to capture

[2] Requests mapped to multiple devices may have a significant impact on the network and hence we have explored this in this work.

[3] Loss is another QoS metric that is mentioned at times but it is not considered here due to packet loss being strictly undesirable in datacenters

the relationship between these microservices. In order to achieve the QoS requirements of these microservices, we need to take into account the data accessed by them and make appropriate data placement decisions (for ex., two interdependent microservices would bene t from accessing data in the same server). Similarly, work ows in storage systems involve tasks at every step which result in data access. This again brings forth the need for considering application relationships and ensuring that data is placed in a manner that aids E2E QoS differentiation. However, this thesis only deals with independent applications, thus reducing the signi cance of data placement scenarios. Finally, we utilize the application requirements mentioned in this section (concerning throughput and latency) to classify applications into different "QoS classes" later in 1.3.1 and introduce new mechanisms wherever required to provide E2E QoS differentiation. However, we rst discuss the issues that pose as obstacles for providing this QoS differentiation.

1.2 Background and Motivation

1.2.1 Network Issues in Remote Access

Almost all of datacenter storage storage today is accessed via the network. Previously this remote storage was comprised majorly of Hard Disk Drives (HDDs), whose storage throughput was signi cantly lesser than the available network link bandwidth, while the storage latency also dominated the network latency. However, storage technologies have grown in the past couple of decades with devices such as Solid State Drives (SSDs) that utilize the Non Volatile Memory Express (NVMe) protocol to drive high storage throughput (ex. 30-40Gbps) while delivering low latency (ex. 50-100µs). Hence only a few of these devices can congest a network link of 100Gbps. The storage access protocol (i.e. NVMe) provides its own notion of QoS differentiation by using the Weighted Round Robin queue arbitration principle on its interfacing queues. However, achieving QoS in the network during a congested state is a dif cult problem to solve as traditional network protocols aim to guarantee fairness to different ows (i.e. applications) in such a scenario. For

example, the widely used Data Center Transmission Control Protocol (DCTCP) would divide the available bandwidth equally amongst all ows during congestion even though these applications may have differing QoS requirements.

Additionally, the latency improvement in storage technology has trended from hundreds of milliseconds to tens of microseconds. This evolution in storage performance has continued in newer storage technologies too, ex. Storage Class Memory (SCM) devices like Intel Optane report low latencies like 5-10s. These faster SCM devices such as Intel Optane Persistent Memory (PM) and the evolving MRAM technology have been making inroads into datacenter storage servers. Storage devices are pooled to form virtual volumes which can be accessed remotely as single volumes. This allows for easy manageability and shared access between multiple hosts. However, access to these volumes via the network can add tens of s to the end-to-end latency due to varying factors (ex. network protocol overhead, existence of network components such as switches, etc). Additionally, with congestion these latencies can be exorbitantly high, thus threatening the goal of QoS differentiation. Finally, PMs support byte addressable mode that also allows them to be accessed as a nonvolatile memory device. Accesses in this mode are typically 1-4 cachelines (similar to memory accesses) and inline (i.e. the hardware thread stalls for the access to be completed). Using remote PM amongst multiple hosts is an attractive proposition to reduce cost but its feasibility may be threatened if the network itself adds latency overhead close to 100x the actual device latency.

1.2.2 High Tail Latencies in Storage Devices

Despite modern-day storage devices like SSDs reporting median latencies in the 10s of s mark, devices that utilize NAND ash architecture (ex. SSDs) to store and retrieve data exhibit high tail latencies. For example, the 95th percentile latency of such a device can extend into a few milliseconds. This is due to the " ash" technology that the SSD backend contains. A ash "cell" can store only 1, 2, 3, or 4 bits of information depending on the type

of technology. These cells need to be programmed to update previously stored information. The high cost involved with this update (due to the slow erasure of a cell followed by the eventual write) results in a no-overwrite policy exhibited by these devices, i.e. updates on data are performed out-of-place while the stale data is marked obsolete. This obsolete data is collected and erased during “garbage collection” which occurs periodically. Additionally, each cell can be erased and rewritten a set number of times, which is known as the “endurance limit” of the device. Hence the writes on these cells need to be spread out using a mechanism called “wear leveling” so that the cells age similarly. These complexities (along with more discussed in other chapters) contribute to the high tail latencies and thus make the access latencies non-deterministic in nature. These high tail latencies can at times even dwarf the network latency. Guaranteeing QoS differentiation depending on the needs of applications takes paramount importance in such scenarios. Additionally, SSDs use multiple levels of queuing inside the device itself which may contribute to the device access latency. These queues (unlike the NVMe interfacing queues) are not QoS aware and hence requests entering these queues are all treated the same. Hence queuing latency inside the device could also be a contributing factor to high device latency.

1.2.3 Need for QoS in the Host End

The end-to-end path of a request starts and ends at the host system as shown in Figure 1.1. These systems themselves may contribute signi cantly to the E2E latency. For example, queuing delays in the host system's send/receive queues can extend by 10s of microseconds even if the network and storage end provide proper QoS treatment. Similarly, processing delays could also cause signi cant overhead. Applications such as remote PM applications require latencies 3 5 s even during congestion. Hence bottlenecks in the host end itself could skew the entire E2E latency in spite of consistent QoS treatment at other points.

Additionally, certain applications can drive huge amounts of data to and from the host

end (ex. FTP applications). These applications may not be latency-sensitive but may interfere with the requirements of other applications if there is no prioritized treatment. For example, an application with a high throughput demand may download large les from the storage server but may not have tight latency constraints. On the other hand, another application (ex. videoconferencing applications) may have high throughput demand coupled with low latency tolerance. In such a scenario, the high throughput demand of both applications may eclipse the latency requirement of the second application. Hence it makes sense for the host end to provide QoS differentiation too in order to address these scenarios.

1.3 Contributions

1.3.1 E2E QoS

As per our discussions till now, we have observed that in order to guarantee QoS differentiation, we need to focus on four major components in a request's E2E path. We call these the ABCs of E2E QoS:

A. The Host End from where the request starts and ends

B. The network path via which the request and response traverses. This includes switches and routers in the path.

C. The Storage End comprising the access protocol and the storage device

In this work, we propose new mechanisms (and also utilize existing techniques) at these four major points to help us provide QoS differentiation. However, we rst utilize the requirements of applications to classify them into QoS classes to aid differentiated treatment at these points. These applications can be broadly classi ed into two different classes of datacenter traf c depending on their QoS requirements: (1) high priority applications and (2) normal priority applications. For example, as shown in Table1.2, applications such as Video-on-demand (ex. YouTube) require High throughput and applications such as Voice

over IP or Videoconferencing require Low latency. These could be classi ed as High priority applications due to their higher demands, while the rest of the applications could be treated as Normal priority applications. Both these classes of applications could be further divided into sub-classes depending on their requirements. For example, in Table1.2, Email, Chat, and Video Streaming are all Normal priority applications. However, they themselves could be classi ed into three different sub-classes, with Video streaming occupying the highest sub-class (due to its Medium to High throughput demand) while Email occupies the lowest class (due to its Low throughput demand).

Figure 1.2: QoS Classi cation

Figure 1.2 shows how we can classify applications according to their QoS requirements. Typically, applications are either throughput-centric or latency-centric (they can be both too, however only one metric is usually tightly bound). Depending on either throughput or latency requirements, they could be classi ed into High or Normal priority as mentioned. Following this, both High priority and Low priorities could be further classi ed into classes (i.e. sub-classes $HP_1 ... HP_n$ for high priority and $NP_1 .... NP_n$ for normal priority). These sub-classes can be differentiated from each other by a speci ed ratio. For example, if the throughput-centric High priority contains 3 further sub-classes HP_1, HP_2 and HP_3, they could be provided differentiated throughput treatment in the ratio of 3:2:1 respectively. This can be extended for Normal Priority applications too. Additionally, datacenter traf c can also consist of QoS-agnostic applications (i.e. applications that do not require QoS

differentiation). These in turn can be clubbed into a single class known as “Best Effort”. As the name suggests, this class receives the lowest priority and it makes way for QoS-sensitive traf c (i.e. High Priority and Normal Priority) as and when required. We now look into our contributions in this thesis and discuss what else could be done to adequately address an application's E2E QoS needs.

1.3.2 Contributions in Different E2E Path Components

As mentioned in 1.3.1, it becomes crucial to navigate bottlenecks present in 4 major points - the host end, the network end, at the storage access protocol level and inside the device. A simple way to tackle issues in the host end for applications requiring low latency could be to cache blocks (or pages) of data in a host-level cache. Hence in Chapter 2 we introduce a novel host memory caching mechanism called FussyCache. FussyCache considers the tradeoff between the cache management cost and the cost involved in accessing low latency devices (ex. SCM devices and datacenter SSDs) in order to make caching decisions. True to its name, it makes a “fuss” about what data is popular enough to store in the host memory cache.

In the network we aim to differentiate between different classes of traf c during a congestion episode while also providing priority to HP classes. We hence modify existing transport protocols used in the datacenter, i.e. DCTCP (Datacenter TCP) and DCQCN (Datacenter Quantized Congestion Noti cation) to propose our new QoS aware solutions known as QTCP and QRDMA respectively. Both these mechanisms utilize a Quality Factor metric that modulates the sender ow rates according to a xed ratio between the QoS classes. This could be applied to both High Priority and Normal Priority classes. However, High Priority classes require further distinction due to their ultra-low latency (or dedicated throughput) requirement. To address this we propose in-network reservation for these classes by reservation of buffer space in the switches. We discuss these mechanisms

in detail (along with their existence in an E2E context) in Chapter 4.[4]

The NVMe protocol proposes the PLM (Predictable Latency Mode) feature to tackle the nondeterministic latencies exhibited by SSDs. However, it does not cater to a multi-host scenario, nor does it consider the QoS classes associated with these hosts. In Chapter 3 we introduce a coordinator module called PLMC (PLM Controller) that arbitrates an application's access to “deterministic” resources according to their QoS classes. Due to the lack of current SSDs with the PLM feature enabled, we also propose a PLMC-like solution called PLMLight.[5] Additionally, we tackle queuing latencies inside the device by extending the existing NVMe level QoS differentiation (i.e. Weighted Round robin queue arbitration) to the in-device queues in Chapter 4. We also coordinate it with the network level and NVMe level QoS differentiation to evaluate E2E QoS differentiation scenarios. In some cases we also explore treating the lowest priority Normal Priority application as a Best Effort class.

1.3.3 E2E Contributions and Outline

Finally, due to the dif culty in creating a real datacenter environment to test our hypotheses, we propose an E2E QoS differentiating tool called NeSt - A QoS Differentiating Networked Storage Simulator. We discuss and evaluate this tool extensively in Chapter 5 where we consider the storage model of the SCM device. We also propose an E2E request tagging mechanism that brings our isolated QoS differentiating components together in Chapter 4.

The rest of this work is outlined as follows: Chapter 2 talks about our proposed Fussy-Cache caching mechanism, followed by Chapter 3 which discusses the PLMC module in detail along with our proposed PLMLight. Chapter 4 talks about our QTCP and QRDMA protocol while also addressing E2E QoS differentiation by combining our proposed modi-

[4] A section of this work was published separately in collaboration with Dr. Joyanta Roy.

[5] Both these works were carried out in collaboration with Dr. Tanaya Roy.

cations in 1.3.2. Following this in Chapter 5 we talk about novel E2E QoS differentiating Networked Storage Simulator - NeSt. We nally conclude this work by talking about the scope of future work brought forth by this thesis in Chapter 6.

CHAPTER 2

FUSSYCACHE: A CACHING MECHANISM FOR EMERGING STORAGE HIERARCHIES

2.1 Overview

Accesses to remote storage devices are in the form of 4KB blocks, which are mapped to the storage backend by logical block addresses (LBA). The latency cost of an E2E access that results in retrieving these blocks of data from the device can extend to milliseconds during a congestion episode. A simple way to tackle this problem for applications requiring low latency could be to cache blocks (or pages) of data in a host-level cache. Caching is a pervasive technique used all throughout the E2E path at different points to reduce latency. Bottlenecks in the network or when the storage device results in high latency values. In such cases, it may make sense to cache data at the host end to reduce device access cost. Hence this chapter proposes a novel caching mechanism, called FussyCache, that differs from the traditional DRAM caching mechanisms that automatically cache a data block when it is requested. Instead, FussyCache evaluates each requested data block for its caching eligibility, and reads ineligible blocks directly from the device each time. We show that the FussyCache performs substantially better than the traditional caching algorithms and its performance increases with the storage device speed. In particular, for the rst generation Intel Optane based storage, FussyCache provides 25-30% reduction in the average access latency as compared to the native caching mechanism such as plain LRU. We also observe close to 15-20% improvement in performance even for a mainstream TLC SSD. Furthermore, the FussyCache design includes two mechanisms that allow for its easy deployment in any environment: (a) a self-monitoring stage that reverts it to a normal LRU when partial caching is not bene cial, and (b) a training phase that automatically tunes the

con gurable parameters for the deployment environment. We explore how FussyCache can be used to reduce average latencies of applications. Later on in our concluding chapter, we discuss how it can be extended for QoS aware treatment.

2.2 Introduction

The emerging nonvolatile memory and storage technologies are beginning to ll the huge gap that traditionally existed between the magnetic disk drive based storage and DRAM based memory. It is now possible to have the storage hierarchies where at least some of the adjacent layers may differ only by an order of magnitude in speed. In such cases, a traditional caching mechanism that blindly caches everything from the lower level is not desirable. Furthermore, at the highest level of the storage hierarchy, the overhead of managing the DRAM cache could be comparable to the read/write latency of the storage device. Thus, it may be bene cial to only cache frequently requested data while serving other requests from the next lower level directly. At the same time, the complexity of the caching mechanism needs to be commensurate with the impact of management overhead.

As an example, the recently released Intel Optane, and particularly the 2nd generation Optane released in 2020 [Mellor (2019)], has a latency of only 10-20 s [Izraelevitz et al. (2019)] to read/write a 4 KB block, which is getting closer to the memory latency, which is in the neighborhood of 1 s.[1] It is also worth noting that a 4KB data transfer size is rather small, and applications often use larger transfer granularity. The gap narrows further with larger transfer sizes since all storage technologies suffer a signi cant initial access latency whose contribution goes down with the size. For example, the current Samsung Evo Plus SSD can achieve read rate of 3 GB/sec and the 2nd generation Optane should also achieve similar rates. Thus the actual time required to read a 4KB block is only about 1.3 s, the rest being overhead. Large transfers from SSDs and Optanes may reduce the gap between

[1] Basic memory accesses are in cachelines but the time to retrieve a large block of data depends on many internal details of DRAM organization.s

memory and storage latency to only a few X.

In this chapter we introduce a novel caching mechanism called “FussyCache” for such systems that is “fussy” about deciding what and whether to cache an item instead of retrieving it from the device each time. In particular, only the items that are deemed suf ciently popular are cached, while others are obtained from the device directly. The mechanism is also self-regulating in that it tracks its own performance and switches over to a simple native mechanism if it is doing worse and switches back when appropriate. Using the available storage system traces, we show that FussyCache for Intel Optane based storage can reduce the average latency by almost 20% compared to the native caching mechanism such as plain LRU. Furthermore, the performance advantage over the more sophisticated caching mechanisms is often larger, not smaller, because of their greater overhead.

With very fast storage, the overhead of caching mechanisms in terms of both computation and memory reads/writes becomes signi cant. For example, a mechanism that requires maintaining substantial amount of information about the access patterns to enable intelligent prefetching or retention could become unattractive unless the additional overhead can be more than compensated by the improvement in the hit rate. Consequently, we mainly use the popular LRU (least-recently used) caching mechanism [O'neil et al. (1993)] as a baseline because of its simplicity and effectiveness in practice. However, we do compare FussyCache against other well-known and effective variants of LRU. We do not believe that a comparison against high overhead or ad hoc mechanisms is warranted.

The outline of the rest of the chapter is as follows. Section 2.3 discusses the related work. Section 2.5 discusses the methodology followed by Section 2.6 which gives an overview regarding the implementation of FussyCache. Section 2.7 then discusses the experimental evaluation. Finally, section 2.8 concludes the chapter.

2.3 Related Work

Caching is an extremely rich area with numerous algorithms; the most popular ones being LRU and its variants such as LFU (least frequently used), LIRS (Low Inter-reference Recency Set) [Jiang and Zhang (2002)], ARC (Adaptive Replacement Cache) [Megiddo and Modha (2003)], ACME [Ar et al. (2002)], and Sequential Adaptive Replacement Cache (SARC)[Gill and Modha (2005)]. Other related algorithms include Adaptive Multi-stream Prefetching (AMP)[Gill and Bathen (2007)], Domino [Bakhshalipour et al. (2018)], Sampled Temporal Memory Streaming (STMS) [Wenisch et al. (2009)], Tombolo [Yang et al. (2016)], etc.

Of these, LRU is the most frequently used algorithm in practice because of its simplicity and performance. Adaptive Replacement Cache (ARC) is a variant of LRU that takes into account both recency of access (like LRU) and frequency of access while evaluating cache candidates. SARC (Sequential Adaptive Replacement Cache) is a caching algorithm that leverages sequentiality so as to make decisions on prefetching. It is similar to ARC on the basis that it utilizes two different factors while making caching decisions but happens to be a more sophisticated algorithm than ARC, thus making it comparatively costlier, performance-wise.

LIRS was another algorithm we considered, however there were certain details in the implementation provided by the authors [Jiang (2016)] that make it very expensive on the memory front. The algorithm creates an in-memory data structure that contains metadata pertaining to the entire LBA range of the workload, which is suitable only for an in-memory workload. In the normal case, the LIRS metadata would itself occupy a huge amount of memory, thereby making it unsuitable and unscalable.

Our criteria for selecting caching algorithms to compare FussyCache with revolved solutions that are comparatively less concerned with data movement and which work well across different types of workloads. This is the reason why LRU and ARC made the cut.

SARC was selected so as to test FussyCache against a prefetching algorithm that leverages sequentiality to make caching decisions. Tombolo could also have been a viable option but its ensemble nature makes it a more complex algorithm than SARC. AMP on the other hand maintains a graph which itself can be costly for workloads that do not exhibit sequentiality. ACME works on the hypothesis that the speed difference between the CPU and the storage device is ever-increasing, thus using machine learning to determine cache candidates. This in turn makes ACME costlier than our chosen alternatives.

2.4 Hit Rate vs. Latency

The key objective of having a cache in the path to any memory/storage device is to reduce the average data access latency, henceforth denoted as L_{avg}. Let L_{lookup} denote the latency to lookup the cache. For simplicity, we also include the small additional overhead of any management operations in case of a hit assuming that this does not involve any further search through the data structure. Let L_{device} denote the device access latency. Also let L_{dm} denote the latency of data management (dm) in the cache when we read in a new item from the device (insertion, any movement of entries, and eviction of some entry). If h is the hit ratio of the cache, then in traditional caching we have:

$$L_{avg} = L_{lookup} + (1 - h) \times (L_{device} + L_{dm}) \tag{2.1}$$

For our mechanism, the relationship is as follows:

$$L_{avg} = L_{lookup} + (1 - h) \times r \times L_{device} + r \times (1 - f(h)) \times L_{dm} + f(h) \times L_{mdm} \times s \tag{2.2}$$

where $f(h)$ is the "fussiness" index, or the probability of not bringing the item into the cache, and L_{mdm} refers to metadata management, or the operations we need to do to record the fact that the item was requested but we did not bring it in. The equation reduces to the previous equation if $f(h) = 0$. We expect that as the hit rate h increases, $f(h)$ will decrease, since the item will tend to stay in the cache.

Although L_{avg} will decrease with increasing h in both eqns (2.1) and (2.2), at the same hit ratio, the latency would be lower for FussyCache if $L_{mdm} < L_{dm}$, i.e., if the metadata management latency is kept small. Beyond an ef cient data structure, this can be done by delegating the metadata operations to a separate thread, as we have done in our implementation.

Henceforth, we shall mostly focus on latency rather than the hit ratio. The results section clearly shows that FussyCache achieves consistently lower latency than other algorithms even though the hit ratio is not always the lowest.

2.5 Implementation Methodology

Our approach consists of splitting the usual memory cache into 2 caches - a data cache (DC) and a dynamic metadata cache (DMC). Both caches are block caches i.e. data is assumed to be accessed in the LBA level, although we will treat LBAs more like chunks. The major difference between DC and DMC is that the latter does not store data corresponding to the block addresses. The DC houses only the identi ed popular data while the DMC stores the frequency corresponding to recently accessed blocks. The DMC also monitors the blocks that have turned popular i.e. blocks whose frequency has crossed a certain threshold value. Newly popular blocks are inserted into the DC along with the data, whereas the blocks evicted from the DC are inserted into the DMC (without the data, and the data is discarded).

DMC also keeps a check on whether the DC's hit rate is decreasing and if so it switches the DC to a traditional native cache, which implements a simple traditional mechanism. This is to enforce the “do-no-harm” idea. There are certain scenarios, as discussed later, where the shuf ing of the items between the DMC and DC is not useful and thus ends up hurting the performance. In such cases, the mechanism automatically reverts to the native scheme. However, the DMC does continue to run in the background so as to detect whether the original mechanism can be brought to the forefront again. The targets of incoming

requests are first checked if they have been marked as popular or not. If they are, their data is cached in the DC. In both cases, the metadata is entered into DMC (for new requests) or updated. Even in the case of reverting back to the traditional LRU, the DMC is utilized by the LRU to bring the data to the cache. This is further explained in Section 2.7.

For the traditional cache, we considered existing caching mechanisms that are known to have high a hit rate and are comparatively less complex as mentioned. LRU was chosen due its being one of the most lightweight and efficient caching solutions. It thus tends to provide the most generalized solution.

The mechanism currently treats both reads and writes identically i.e. it does not distinguish between read popular data and write popular data. This is reasonable when the read and write latencies do not differ much; however, for storage devices with substantial difference in read and write speeds, it would be useful to weight the reads and writes differently. For example, the current high end SSDs and Optane drives do not show much difference in their read/write latencies. The phase change memory (PCM) based devices [Gallo and Sebastian (2020)], when available, are expected to have significant differences in read and write speed because of the basic nature of the operations (write performed by melting of material and its cooling, whereas read is simply a check on the conductivity properties).

2.5.1 FussyCache Parameters

Our algorithm has the following internal parameters:

1. freqThreshold: This parameter determines whether a certain block is popular or not. Once a certain block has been accessed more than freqThreshold number of times, it is declared to be popular.

2. accessThreshold: This parameter is used for two functionalities - hit-rate derivative calculation and population of the DC. The former deals with detecting whether a traditional mechanism is desired over the current mechanism for the given workload

while the latter is applicable to when the DMC performs a check on all its contents in order to identify new candidates for DC. Every time the number of processed requests crosses accessThreshold, the two mentioned events are triggered.

3. dmcsize: This determines the size of the DMC. It is calculated during the warmup phase.

4. hrCount: hrCount is used to determine the number of consecutive times the DMC allows the hitrate to deteriorate. Beyond which it reverts to a traditional LRU cache. It doubles every time a switch is made between the caches. Our implementation starts off with a value of 5 i.e. it checks every accessThreshold number of requests whether the hit-rate has deteriorated for 5 consecutive times before it can make the first switch.

5. sleepTimer: This parameter decides how long the DMC thread sleeps and is dependant on the accessThreshold parameter.

2.5.2 Dynamic Metadata Cache

The contents of the Dynamic Metadata Cache (DMC) are characterized by block numbers, their frequency value (defining the number of times each of these blocks have been accessed) and a boolean value which indicates whether the block is popular or not. Pointers to the popular blocks are added to a data structure which houses the pointers to all popular blocks. No data corresponding to the blocks is a part of the DMC. However, data corresponding to the requests directed at the DMC is fetched from the next layer of storage while in the meantime and its presence in the DMC is checked. If it is already present, the frequency of the corresponding block is incremented, else the block is inserted into the cache along with a frequency of 1 as this is the first time it has been accessed in the given recent past. In short, the DMC consists of unpopular blocks that have been accessed recently. The

Algorithm 1 DYNAMIC METADATA CACHE

```
hitrate, hrMeasure, DC.accesses, LRU ← 0
while true do
    sleep ( sleepTimer )
    if accesses ≥ accessThreshold then
        if LRU then
            if popularBlocks.count > DMC.count / 2 then
                hrCount ← hrCount / 2
                hitrate, hrMeasure, DC.accesses, LRU ← 0
            end if
            if hrMeasure ≥ hrCount then
                hrCount ← hrCount / 2
                hitrate, hrMeasure, DC.accesses, LRU ← 0
            end if
        else
            for i ← 1 to popularBlocks.count do
                Enqueue(popularBlocks[i])
            end for
            hrTemp ← hitrate
            hitrate ← DC.accesses / accesses
            if hrTemp ≥ hitrate then
                hrMeasure ← hrMeasure + 1
                else hrMeasure ← 0
            end if
            if hrMeasure ≥ hrCount then
                hrCount ← hrCount / 2
                hitrate, hrMeasure, DC.accesses ← 0
                LRU ← 1
            end if
        end if
    end if
end while
```

frequency of these recently accessed blocks is observed so as to see if any of them turn popular. Old, least accessed blocks in the DMC make way for newly accessed blocks.

While serving requests, the DMC performs two other operations for every accessThreshold number of accesses:

- Identification: The DMC checks the frequency of popular blocks present in it and their data is fetched from the storage device and inserted into the DC.

- Cache Change: The DMC also keeps a check on the hit-rate of the DC. It checks if the hit-rate at the current check is smaller than the one during the previous check. If this check is satis ed for *hrCount* number of times, the entire caching mechanism shifts to a traditional LRU i.e. all DC misses are brought into the DC. The DMC keeps checking the popularity of incoming blocks so as to observe whether the hit-rate to the DC increases for *hrCount* number of times. If it does, it switches back to the previous methodology. Even if it doesn't it still switches back to the original mechanism after the updated *hrCount* number of times. This is because the hitrate may have fallen because of a change in workload and so FussyCache is introduced again to see if it makes any gains. This switching back and forth does not result in an exorbitant cost because as the parameter *hrCount* is doubled during a switch, the probability of switching back in the near future decreases.

The given pseudocode for the DMC in Algorithm 1 assumes that the DMC (implemented as an array of nodes) has its *count* data member that measures the number of blocks present in it and the DC cache has a data member calculating the number of accesses to it. *hitrate* and *hrMeasure* measure the hitrate of the DC and the count of the number of intervals for which the hitrate has deteriorated respectively. *isLRU* stores whether a switch to LRU has been made. Finally, *accesses* keeps a measure of the total number of accesses that have been processed till now. A separate data structure *popularBlock* contains pointers to popular blocks in the DMC. Whenever a block in the DMC crosses the *freqThreshold*, a pointer to it is added to *popularBlocks*. After every *accessThreshold* number of requests, all the *popularBlocks* are added to the DC.

2.5.3 Data Cache

The Data Cache (DC) is treated like a traditional LRU cache in terms of its operation. Incoming blocks are placed at the head of the LRU queue and evictions happen only on the basis of the least recently used block. However, not all blocks that are requested are inserted into the DC. If a certain block is not identi ed as popular by the DMC, then it

remains in the DMC until it is accessed *freqThreshold* number of times. Evictions from the DC are not treated like a conventional LRU. If it is a block that has been written to, then it is written back to the storage device. However, irrespective of a read or write, details of an evicted block from the DC is stored in the DMC. This gives it a chance to be popular again because a block that was popular in the recent past may turn popular again in the near future.

The DC is initially populated instead of starting empty. It acts like a training period during which it initially treats all requests as random and unpopular, hence bringing them into the DMC first before the DC. As these unpopular blocks are requested for multiple number of times, their popularity in the DMC increases. Once these DMC candidates cross the *freqThreshold*, they are shifted to the DC. This phase is introduced just so that all data initially is not treated as unpopular i.e. all requests are considered to be random in nature. This leads us to the *Warmup* phase in Section 2.5.4.

2.5.4 Warmup

As mentioned before, the *Warmup* phase makes sure that FC does that start at time 0. It gives the mechanism a head start so as to populate the DC and tackle the mentioned problem. In our implementation we have used the first two thousand accesses for every trace used for this warmup phase. The *Warmup* phase is also used for calculating the parameter values. The *freqThreshold* and *accessThreshold* parameters are calculated by a combination of binary search for the two parameters in question. It tunes *freqThreshold* using binary search while keeping *accessThreshold* constant. It then fixes the *freqThreshold* and tunes the value of *accessThreshold*. This process is continued until it converges. The value of *freqThreshold* tends to be low, approximately 3-4. The *dmcsize* parameter which, as the name suggests, is the size of the DMC, is calculated after the DC has been populated by checking how many items reside in DMC during the *Warmup* period. This tends to be approximately half the size of the DC. Note that in most situations the DMC size will be

Figure 2.1: Use of multithreading

insigni cant as compared to the data cache size since the DMC stores only the metadata associated with the data blocks.

2.6 Implementation of FussyCache

2.6.1 Experimental Setup

We implemented our scheme on an Inspur NF8250m5 server with a dual Intel Xeon processor (i.e. 12x2 cores). The storage devices consisted of (a) an Intel Optane DC SSD, (b) an Intel SSD DC P4510. The two different devices were used so as to observe the impact of different device speeds and access latencies on the performance.

Managing two caches (along with their synchronization) in the memory can increase latency because every miss to the DC results in several operations, such as fetching the requested block of data from the storage device, populating the DMC with the metadata pertaining to this block, and periodically checking the entire DMC contents. To address this, we use two different threads, each running on a different core, as shown in Figure 2.1. We also use a third thread to coordinate event handling between DC and DMC, the events being the operations discussed before.

Core 1 - Thread 1 This thread carries out two operations, namely identification of popular blocks in DMC for transfer to the DC, and switch-over from FussyCache to the traditional cache, as driven by the accessThreshold parameter. This thread is normally sleeping and is awakened every time the number of accesses has crossed the parameter accessThreshold. After carrying out the mentioned operations, it goes back to sleep. This keeps on repeating until it is signaled by Thread 2 to stop and exit.

Core 2 - Thread 2 This thread handles all incoming requests. The requested block is first checked for presence in DC (via a hash table for fast lookup). If it is present, it operates the DC exactly like an LRU i.e. the requested block is placed at the top of the LRU list. If it is not present, it branches out into two threads:

- Thread 2A : This thread handles device IO. In case of read, it fetches the data from the device, whereas in case of a write the data is directly written on the device without going through the DRAM cache.
- Thread 2B : This thread is used to insert the requested block-descriptor into the DMC. It first checks whether it is already present in the DMC (again, using another hash table).

After processing all requests, Thread 2 signals Thread 1 to exit before exiting itself. This parallelism is confined to the cache itself and does not affect the applications. Among LRU, ARC and SARC, only SARC is implemented in a multi threaded manner. As FussyCache performs all operations of the DC (which is essentially an LRU) on a single thread, the implementations of LRU and ARC are kept as single threaded. Even when FussyCache reverts to a traditional LRU, the latter carries out all operations on a single thread.

2.6.2 Benchmarks

For evaluating and comparing our proposed mechanism with other existing methodologies, we used a wide variety of workloads such as Systor '17 traces [Repository (2020)]

Table 2.1: Average Request Size in KB

Workload					
SYSTOR	FIU	DBLOG	DBTABLE	VDA	SWBUILD
9	8	18	18	457	23

(utilized by Lee et.al [Lee et al. (2017)] in their work and published in Systor'17), FIU Web Research trace [SNIA (2008)] and also workloads provided in SPECSFS 2014 benchmark suite [SPEC (2014)]. These workloads were chosen so as to cover the gamut of workload characteristics and to also test workloads with different read:write ratios so as to consolidate our hypothesis that FussyCache works well as a generalized caching solution.

The Systor'17 traces are actually a read oriented Virtual Desktop Interface (VDI) workload divided between 7 LUNs, of which we have used 2 for our experiments (LUN0 and LUN1). Accesses ranged from 1.5-2 million per hour, with an average request size of 9KB. The read:write ratio was 7:3 with almost 80k distinct blocks being accessed in the workloads.

The FIU Web Research workload is a read only workload which is a trace of a Web-based management workload centering 10 FIU research projects using an Apache web server. The average request size for this workload is 8KB with the number of accesses ranging between 1.2-1.5 million per hour.

From the SPECSFS suite, we used the 2 database workloads included (DBTABLE and DBLOG) and also 2 other workloads - SWBUILD (software build workload) and VDA (Video Data Acquisition). This was done so as to test our mechanism against standard benchmarks.

The database table workload has a read:write ratio of 4:1 while the software build workload has a read:write ratio of 1:4. Finally, both Database Log and VDA are write only workloads. Altogether, the set of workloads considered here spans a large range with respect to read:write ratios and access characteristics. The SPECSFS workloads have an LBA space

(a) Average Read Latency for ENVM with cache size as 10%

(b) Average Write Latency for ENVM with cache size as 10%

(c) Average Read Latency for ENVM with a cache size as 5%

(d) Average Write Latency for ENVM with cache size as 5%

(e) Average Read Latency for ENVM with cache size as 2.5%

(f) Average Write Latency for ENVM with cache size as 2.5%

Figure 2.2: Average access latencies (in μs) for ENVM

of about 10-20k blocks being accessed.

During the warmup phase, the size of the DMC is calculated so as to make sure that the blocks get an appropriate amount of time to turn “popular” before being evicted from the cache.

2.7 Results

As mentioned in Section 2.6.1, we have two different storage devices with varying characteristics so as to test our mechanism in two different setups. The Intel Optane reports access latencies ranging from 10-20μs while the Intel TLC SSD reports access latencies in the range of 30-110μs [Chiappetta (2018)]. Let us call the former ENVM (Emerging Non Volatile Memory) and the latter LL2SSD (Low Latency SSD).

ENVM

For theENVM setup, we test the different caching algorithms on three different cache sizes - 10%, 5% and 2.5% of the LBA space. We have chosen these speci c percentages so as to observe the behavior of FussyCache(compared to the other algorithms) when the cache decreases from a large cache to a noticeably small one. A cache size of 10% for a workload whose LBA space ranges around a couple million blocks may result in a cache that can house a maximum of 80k blocks (which translates to almost 1GB memory space if we assume the block size to be 4KB) while a cache size of 2.5% would consume only 200 megabytes of memory. Hence it is interesting to observe how caching algorithms perform in these two contrasting situations and also when the cache size lies somewhere in between the two ends.

In this setup we can observe that FussyCache performs better than the other mechanisms for all workloads, both for reads and writes. This behavior is re ected for both the cache sizes that are tested. For reads in the Systor workload (LUN0) with a larger cache, FussyCache performs almost 16% better than LRU which in turn performs the best among the other algorithms in question. Concerning the write latency for the same workload, FussyCache performs almost 10% better than LRU. Moving on to the other Systor trace (LUN1), we notice that similar gains are observed. FussyCache averages a 10-20% performance increase compared to the other algorithms for both reads and writes. When the cache size is reduced to 2.5%, we see that FussyCache averages at a 15% percentage increase in performance compared to LRU which does better than ARC and SARC. For a cache size of 5% no changes in behavior are observed i.e. FussyCache performs the best.

For the FIU Web Search workload (which is a read only workload), we can see in Figure 2.2a that only ARC and SARC perform better than LRU but FussyCache performs the best. This is due to the fact that FussyCache leverages the frequently accessed blocks well but at the same time serves the unpopular blocks from the backend device. ARC does better

than LRU and SARC because it also considers frequency. But existing caching solutions take into account only the frontend device where the cache itself resides and cache every item accessed indiscriminately. They are not guided by the backend device's capabilities. FussyCache performs almost 15% better than ARC and close to 25% better than LRU for this workload. In the case of the smaller cache of size 2.5%, FussyCache makes almost 13% gains compared to the other mechanisms. For the cache size 5%, FussyCache performs almost 20% better than SARC, whose performance is very much similar to ARC's.

FussyCache also shows similar behavior for the Database Table and Database Log workload. For DBTable, the gains are in the range of 5-15% for the reads and 5-10% for the writes. For DBLog, which is a write only workload, FussyCache does considerably better than the other algorithms. It performs better than both LRU and SARC by almost 18% while doing better than ARC by 25%. In the case of the DBLog workload, we see reducing the cache size to 2.5% results in all the caches reporting similar performance. But even then FussyCache edges out the other mechanisms, albeit marginally. A cache size of 5% re ects behavior that is close to being identical to that of the largest cache size (10%), with almost 10% improvement being observed.

The VDA workload (a write only workload) is interesting due to its highly sequential nature. The large request size seen in Table I also shows that a prefetching algorithm that accounts for sequentiality in workloads may do well here. And that is mirrored in the results observed in Figure 2.2b. SARC performs better than LRU and ARC by a huge margin due to this aspect. However, FussyCache wins in this case too as recently accessed blocks have a propensity of getting accessed again in this workload i.e. it has an inclination towards requesting the same starting block with varying request sizes. Hence it outperforms ARC by almost 30% and LRU by about 20%. It beats SARC marginally in spite of its simplicity since SARC is built for workloads such as VDA which exhibit such high sequentiality. Even for this workload FussyCache does better than the three algorithms for a smaller cache of size 2.5% of the LBA space, although SARC comes very close to matching that

(a) Average Read Latency for L2S2D

(b) Average Write Latency for L2S2D

(c) Hit Rate for Systor LUN0

Figure 2.3: Average access latencies (s) for L2S2D and Hit Rate for Systor LUN0

performance due to the aforementioned reasons. Even in the case of the cache size being 5%, SARC and FussyCache perform almost identically.

For the Software Build workload, FussyCache executes the Cache Change phase and reverts to an LRU. However, for this particular setup, it performs better than the other mechanisms in question because FussyCache utilizes FussyCache utilizes the DMC so as to bring data from the backend to the cache. Thus using its own inbuilt second cache to improve a traditional LRU. The improvement seen in this case ranges from 15-25%. This behavior is mirrored even when the cache size is reduced to 5% and 2.5%, with almost identical differences. For the smallest cache size we can see that almost 20% improvement in performance is exhibited by FussyCache compared to the LRU which performs the best compared to ARC and SARC.

Looking at the results for FussyCache tested across different workloads (which cover different read and write percentages while also encompassing different characteristics), we can say it performs well even when the cache size is varied. In case it identi es that it is not doing a good enough job (as in the case of the Software Build workload), it reverts to an LRU with the DMC helping out, thus making it a two-pronged attack to the given problem. This led us to perform the same tests for the same workloads but with a selected cache size of 10% for the second setup as it makes sense to assume that the behavior will be echoed for varying cache sizes too. The results for the second setup are in the next subsection.

L2S2D

In this setup we utilize the TLC SSD as our backend storage device. As FussyCache is ideal for faster devices, we show that in this setup, even though FussyCache does end up doing better, the gains are reduced due to the difference in access latencies being much more pronounced than in the Intel Optane. As mentioned before, the cache size we have used for this setup is 10% of the LBA space accessed by the workload.

We can see that even in such a setup FussyCache performs well in most cases. For example, in the VDA workload, as seen in Figure 2.2f, FussyCache performs better than LRU and ARC by almost 17% and reports latency values comparable to SARC. It makes gains over ARC by almost 15% and over LRU by 17%.

For the Systor trace (LUN0), we notice that FussyCache performs better than LRU both for reads and writes. The performance improvement is almost 8% compared to LRU for both reads and writes. LRU performs the best among the other three algorithms. Looking at LUN1 trace, we see all the caching solutions in question perform comparably to each other with Fussy Cache performing still outperforming the other three for both reads and writes.

The FIU trace re ects the same observations as before with FussyCache doing better than LRU by about 7%. This proves that using FussyCache on lower latency SSDs can be an effective solution too as we can see that across different workloads FussyCache performs almost as well as widely used caching mechanisms and even does better in some cases. This is a signi cant result since we expect that future SSD offerings will generally provide even lower latencies than the current ones.

Looking at the Database Table and Database Log workloads, we can observe that similar to the reported performances for NVM devices, all four of the caching mechanisms behave in a similar fashion. For Database Table, the read performance between all four caching mechanisms only differs by a maximum of 6% and for writes it is close to 8%. For

Database Log however, LRU outperforms all the other mechanisms. It makes considerable gains over ARC and SARC but differs from FussyCache by only a couple of microseconds.

For the Software Build workload, FussyCache again reverts to a traditional LRU but in this case the gains from the DMC acting as a conduit between the LRU and the backend is not as pronounced due to the slower backend device. However, even in this case, it performs almost identical to LRU and the other two algorithms show similar behavior too.

Evaluating the results for the two setups, we can observe that FussyCache performs better as the difference in access latencies between the frontend memory and the next storage hierarchy is bridged. This is magni ed in Setup 1 which involves the Intel Optane. Even though the difference is not as stark for the second setup, it is still visible and con rms our hypothesis that as devices grow faster, caching algorithms need to adapt too.

Hit Rate Comparison

Even though this chapter talks earlier about how the relationship between hit rate and latency is not as direct when the device in the next layer of the storage hierarchy gets faster, we still compare the hit rates of the other caching mechanisms in question with FussyCache. We show that FussyCache's reported hit rate numbers are comparable to the other algorithms.

Figure 2.4: Comparison of Hit Rates

Figure 2.5: CPU Utilization

We rst look closely at a particular workload, varying the cache sizes for it to observe the difference. Followed by this we compare all the hitrates for the mechanisms tested with the mentioned workloads, using the cache size as 10% of the number of LBAs being accessed.

In Figure 2.3c, we observe how the hit rate varies for LRU, ARC, SARC and Fussy-Cache as we increase the cache from 2.5% of the LBA space to a maximum of 10%. We test this with the Systor trace (LUN0). We notice that for the smallest cache size Fussy Cache does the best followed suit by LRU. This behavior remains unchanged even as we increase the cache size to 5% of the LBA space. LRU starts to perform better than Fussy-Cache as we cross the 7.5% mark but even then FussyCache does not fall behind the other mechanisms and does a decent job. At the 10% mark FussyCache performs better than SARC and ARC, but falls slightly short of LRU.

In Figure 2.4, we compare the hit rates across the different workloads and see that even in this graph we notice the same behavior i.e. FussyCache's hit rate numbers are comparable to the other mechanisms across all the workloads and even outperforming most in all the cases.

We argue that it is acceptable for certain cache misses to occur if that results in only popular blocks being residents of the caches. A particular block may be accessed just twice

or thrice throughout amongst thousands of accessed but that may not make it a popular block. Thus misses to these blocks may in turn slightly affect the hit rate but it helps in the overall performance in the long run. Hence we can say that FussyCache provides a generalized solution across different workloads with different characteristics while performing better than standard caching solutions at hand. It makes sense in an environment that employs an ENVM or faster device in the backend but even stands its ground in other environments.

In Figure 2.5, we can see that the CPU utilization for FC is close to that of ARC. With the advent of systems that have multiple CPUs and numerous cores, it is possible to accommodate caching mechanisms such as FC. It is also to be noted that the usage of multiple threads contributes to the CPU utilization in SARC, hence making it more computationally expensive. However, as stated earlier, the LRU cache in FC (implemented as the DC) performs all of its functionalities on a single thread, similar to a traditional LRU. In the case of FC, tuning the accessThreshold parameter ensures that the DMC thread wakes up only when newly popular blocks are to be re-assessed.

2.8 Conclusions and Discussions

In this chapter, we make the case that with the emerging very fast storage devices, the traditional view of caching where all data read from the device is automatically cached needs to be rethought. In particular, cache blocks that are not very popular are better served directly from the device. The key challenge in implementing this idea is to separate the popular from semi-popular blocks in a light-weight manner. The proposed FussyCache solution provides such an approach.

We have compared FussyCache with three widely used existing caching solutions - LRU, ARC and SARC over five different workloads and shown that it performs almost 25-30% better than traditional caching mechanisms for emerging technologies such as Intel Optane. Although the proposed scheme is expected to be less competitive as the de-

vice speed goes down, we show that it performs competitively with fast SSDs, and only somewhat worse than traditional caching with slow SSDs. An important component of FussyCache is to ensure that it “does no harm”. This is done by its “Cache Change” feature which monitors the FussyCache's partial caching performance and reverts to traditional full caching when needed.

We note that we have tested FussyCache for applications where the requested blocks are directly stored in application's private space in their entirety. This includes storage arrays, database systems, map-reduce applications, etc. In other cases, such as the traditional OS page cache, FussyCache may experience a few misses (de ned by the internal parameter “freqThreshold”) before it caches the block. This may have a detrimental effect in some case while being bene cial in others.

Although we did not explicitly study the endurance impact of FussyCache in this chapter, the endurance should not be adversely affected by much since only the less unpopular writes will go directly to the device; the popular items that receive many writes will still be updated while in DRAM.

2.9 Future Work

FussyCache sits in isolation without coordinating with the QoS requirements of different applications even though it helps in reducing latency for application. I propose a FussyCache v2.0 that has the following changes to accommodate the E2E QoS requirements.

- FussyCache v2.0 sits on host end and makes caching decisions based on E2E Latency. Ultra Low Latency devices such as PM make it an ideal candidate for FussyCache usage.
- The cache can be divided between all four QoS classes i.e. Urgent, High, Medium, Low classes.

- Throughput sensitive applications get less cache space while applications accessing fastest storage tier gets the most cache space i.e. memory traf c
- The DMC divides cache space dynamically between the QoS classes.
- Memory applications received a reserved amount of cache space. The memory overhead for this is low due to small request sizes (128-256 bytes).
- The remaining cache space is divided between the High, Medium and Low classes and the hit rate for these classes is used to determine their cache boundaries.

CHAPTER 3

PLMC: TAIL LATENCY CONTROL MECHANISM FOR NVME SSDS IN SHARED ENVIRONMENT

3.1 Overview

Solid-State Drives involve a complex set of management activities in the background, resulting in unpredictable delays and occasional extended access latencies. However, there is an increasing demand for deterministic I/O access latency in many environments. This demand has prompted a new feature in the NVMe storage access protocol called Predictable Latency Mode (PLM), which is another way to tight control the SSD access latency's tails. This chapter presents the first study of the PLM feature in a single-host environment and its extension to multi-host settings. We propose a PLM Coordinator (PLMC) that regulates access to the PLM of a shared SSD device based on the hosts' traffic characteristics. Our simulation experiments show that the proposed PLMC with a simple traffic prediction can achieve 31% improvement than without a coordinator on the 90%-tail latency values.

Popular applications such as social media platforms require low storage access latency and therefore strict control over the tail of the storage latency is required to provide a good quality of user experience[ ]Xu et al. (2013); DeCandia et al. (2007); Dean and Barroso (2013)]. SSDs using the NVMe (Non-Volatile Memory Express) interface are rapidly becoming standard replacing conventional interfaces such as SATA [Micheloni et al. (2013); Cobb and Huffman (2012); Strass (2018); Huffman (2012)]. The NVMe interface provides several advanced features, including the notion of NVM-sets [NVMe (2019)]. SSDs need to perform many background management operations to deal with the idiosyncrasies of the underlying flash technology, the most time-consuming ones being garbage collection and wear leveling. These operations can increase the access latency's tail from its nominal

value (100 us) up to several milliseconds. Figure 3.1 illustrates the latency distribution of a recent Intel NAND SSD device in an experiment with a workload of 70%-read and 30%-write 4KB requests [Intel (2020)]. As the results show, the 99%-latency can exceed 1 ms due to different background operations and their interference with user I/O accesses.

NVMe v1.4, the latest stable version of NVMe speci cation, has introduced the Predictable Latency Mode (PLM) to provide control over the latency tail [NVMe (2019)] at the granularity of NVM-sets. This feature regulates the periods or time windows over which the SSD controller is allowed to perform the management operations, known as non-deterministic windows or NDWin. Accordingly, during other periods, known as deterministic windows or DTWin, NVM-sets will not experience any management operations. Therefore, IO operations performed during NDWin will result in long latency. Thus, by mirroring the data across multiple (two or more) NVM sets (on different SSDs) and ensuring that at least one of them is always in the DTWin state, it is possible to ensure that a deterministic access is always available.

In this chapter, we explore, for the rst time, how the PLM feature of NVMe v1.4 can be used in a multi-host environment where several applications compete for the limited DTWin budget, i.e., there is a limitation for the total number of IO requests that can be performed entirely during the DTWin period. This happens if (a) several applications have tight tail-latency requirements and thus need to take advantage of the PLM, and (b) the data accessed by these applications is located on the same SSD device. In case all these applications run on the same host, the host itself can regulate their share of the PLM based on the available DTWin counts, i.e., the number of IO's that can be performed during DTWin. However, when the applications run on different hosts, a more complex coordination is required. Typical data center environment concentrates storage in a small number of “storage servers” which are accessed by all the hosts; therefore, the scenario of applications resident on different hosts sharing data on a single SSD is quite common. Typical examples of such shared access include traditional databases, semi-structured key-value stores, or document

stores containing documents, images, videos, etc. Low read latency is one of the major challenge to achieve for these applications.

The existing PLM mechanism, as currently de ned, implicitly assumes access by a single host, and focuses only on serving the reads. We intend to extend the PLM mechanism for shared hosts running read workloads in this chapter to achieve low read latency. Therefore, write requests are not tackled and out of scope of this work. To the best of our knowledge, there is no other comparable mechanism in the literature at this point.

We regulate access to shared SSD data by de ning a PLM Coordinator(PLMC) that interacts with the hosts and allocates a suitable PLM (DTWin "count") budget to each. The PLMC resides in the control path of the hosts and hence does not interfere with the data path. Also, the mechanism is de ned such that it does not disturb the existing NVMe related protocols and is thus easily implementable. In particular, the PLMC, residing locally at the storage server, registers itself with the device as another NVMe host. Since the PLM capability is very new and still not available in any commercial SSDs, we evaluate it both through a very comprehensive SSD simulator and through an approximate emulation on a system that has multiple SSDs of the same type and some persistent memory.

We evaluate our proposed PLMC by considering three different QoS classes in both simulation and emulation. The PLMC predicts traf c of different classes based on a simple mechanism and uses it to allocate the DTWin "counts". In order to study how well the PLMC can do, we also determine the precise traf c from the host side. Using this, we show that the allocation based on the PLMC predictor can achieve up to 31% improvement in serving the highest QoS class compared to serving the same application in traditional PLM, in spite of very high burstiness of the traf c. The detailed analysis of the PLM feature also provides several insights that we believe will be highly valuable in improving the PLM feature and achieving a higher level of determinism in IO latencies. We also propose a PLMC-like solution called PLMLight that segregates writes on a separate device to reduce the chances of a write induced nondeterministic latency. These two works were

Figure 3.1: Latency Distribution for Intel DC P4610 SSD

done in collaboration with Dr. Tanaya Roy.

The outline of the chapter is as follows. Section 3.2 provides an in-depth discussion of the PLM feature as speci ed in NVMe1.4, and its limitations. Section 3.3 then discusses the design of PLMC to coordinate access to shared data from multiple hosts. Section 3.4 discusses PLMC allocation policies in a shared environment and their analysis. Section 3.5 provides some details of our simulation and emulation results. Finally, section 3.8 concludes the discussion.

3.2 Background

The essential characteristic of the ash technology used by SSDs is the need to do out-of-place updates of data and limited endurance. These characteristics require time-consuming maintenance operations such as consolidation of live data, marking of stale data (garbage collection), wear leveling, and block erasures.

SSDs internally pack a lot of complexity and require an internal processor to perform management activities and are usually implemented in rmware (known as Flash Translation Layer) (FTL) [Eshghi and Micheloni (2013); Micheloni and Crippa (2017)]. The

primary role of FTL is to hide the complexities of erasure operation [Bednar and Katos (2011)], out-of-place writes [Abraham et al. (2010)], address translation [Ma et al. (2014); Liu et al. (2016)], wear-leveling [Alsalibi et al. (2018); Kim and Kwak (2019)], and garbage collection [Cai et al. (2017); Chang et al. (2004); Gal and Toledo (2005); Tavakkol et al. (2018)]. These activities and also they interfere with each other can extend the access latency's tail from its nominal value (100 s) into several milliseconds [Kim et al. (2019); Tavakkol et al. (2018)]. For example, Figure 3.1 shows the latency distribution for a recent Intel SSD [Intel (2020)]. It is seen that even 99% latency can exceed 1 ms. Another critical characteristic of ash technology is the limited endurance, i.e., a limit on erasure operation performing on a block before becoming error-prone [Abraham et al. (2010)]. Inaccurate assessment of SSDs can jeopardise SSDs performance and endurance [Mohan et al. (2010); Hu et al. (2011); Roy and Kant (2020); Agrawal et al. (2008)].

3.2.1 NVMe and Deterministic Latency Feature

NVMe v1.4 de nes several concepts that are relevant for PLM. An SSD can be divided into multiple NVM-sets. As stated earlier, PLM operates at the granularity of NVM-sets. An NVM-set can even be divided into multiple “namespaces”, but here we also assume one namespace per NVM-set.

The PLM mechanism cycles an NVM-set between the “DTWin” and “NDWin” time windows, such that all background activities are concentrated only during NDWin period. The DTWin is de ned by two attributes: (a) a prede ned time limit and (b) a prede ned limit on the number of the read and write operations that can perform on an NVM-set. The NVM-set may transition to NDWin if either of these limits is exceeded. The DTWin attributes introduced in NVMe v1.4 spec are listed in Table 3.1. C_{RD} and C_{WR} are collectively mentioned as DTWin “counts” (DC) in this chapter. Write operations to the NVM-set during DTWin are held in persistent memory (e.g., battery-backed DRAM or NVRAM) inside the device. It helps to complete the writes fast and minimize the impact of writes on

the read latency. The NVM-set may transition to the NDWin earlier than the speci ed duration (time or number of IOs) – this happens if there is either an emergency management operation required or the persistent memory buffer is full. The amount of time spent by the NVM-set in NDWin state depends on the necessary background operations for management activity and will be expected to increase with write intensity.

The PLM feature is currently designed primarily to cater to reads since the determinism for read IO latencies is generally more critical than for writes. To achieve the deterministic latency, the host needs to obey the predictable latency operating rules. The device posts the DTWin related status for an NVM-set into a “log-page” accessible to the host. As stated earlier, to provide continuous access to DTWin mode to an application, we need two (or more) different NVM-sets in different endurance sets with identical data. We call this an NVM-set pool, and the NVM-set(s) from this pool currently in DTWin mode as DT-active. By ensuring that all NVM-sets in a pool are never in the NDWin state simultaneously, an application can always read data in the DTWin mode from one of the DT-active NVM-set. With a write (or update), it is necessary to keep the data up-to-date at NVM-set pool granularity, which means that all writes must be sent to all NVM-sets in that pool. However, the writes need not proceed to the ash immediately and may be held in the nonvolatile buffer of the FTL. The PLM speci cally avoids any writes to the device during the DTWin period to ensure that the read latency is not affected.

3.2.2 Predictable Latency Mode (PLM) Operations

Although the FTL possesses the necessary knowledge to decide when management activities are needed for an NVM-set, the host should control the DTWin/NDWin transition based on the applications' needs. The PLM mode tries to achieve a compromise, which is more complicated than a clean device-controlled or host-controlled mechanism could be. In the emerging host-based FTLs (e.g., Open channel SSD [Bjørling et al. (2017))], the host can control the transitions more directly. Since a host could use multiple NVM-sets

Table 3.1: DTWin attributes involved in PLM

Attribute Name	Symbol	De nition
DTWin Reads Typical	C_{RD}	Typical no. of 4K random reads that may be performed in the DTWin
DTWin Writes Typical	C_{WR}	Typical number of writes that may be performed in the DTWin
DTWin Time Maximum	T_{DW}	Maximum time in ms that the NVM-set is able to remain in a DTWin before entering a NDwin
NDWIN Time Minimum High	T_{NDWH}	Minimum time in ms that the NVM-set needs to remain in the NDWin before entering a DTWin. This is the time necessary to prepare for remaining in the Deterministic Window for DTWin Time Maximum
NDWIN Time Minimum Low	T_{NDWL}	Indicates the minimum time in ms that the NVM-set needs to remain in the NDWin before entering a DTWin. This is regardless of the amount of time spent in the previous DTWin
DTWin Reads Estimate	E_{RD}	Estimate of the number of 4K random reads remaining in the current DTWin, if applicable. This value decrements from DTWin Reads Typical to 0h based on host read activity and operating conditions
DTWin Writes Estimate	E_{WR}	Estimate of the no. of writes in units of the Optimal Write Size remaining in the current DTWin, if applicable. This value decrements from DTWin Writes Typical to 0h based on host write activity and operating conditions
DTWin Time Estimate	E_T	Estimate of the time in ms remaining in the current Deterministic Window

simultaneously, they could be at different points in their DTWin/NDWin cycles. Therefore, it is essential that the host is able to in uence DTWin/NDWin duration to synchronize multiple NVM-sets in an NVM-set pool. Thus, we need a NVM-set coordinator (NSC), which can communicate with the host by logging appropriate status information in the completion status to a “log page” per NVM-set. PLM's current speci cation puts the regular transition control with the host and the forced transition by the NSC (in case the host does not be-

have) to the device. Thus, the host de nes the beginning of DTWin and NDWin periods by making requests to the NSC via a “set-feature” command.

Along with the NVMe v1.4 de ned DTWin attributes, we have added another attribute, ratio ($_{NH}$ { $_{DH}$), that de nes the rate of the management activity relative to the regular usage of the NVM-set. With all these parameters, the key idea is for the host to estimate E_{RD} , E_{WR} , E_T of an NVM-set before requesting the NSC to transition to NDWin for a speci ed period. It can also adjust the duration across multiple NVM-sets to synchronize them better. However, the hard limits of these adjustments are set by the NSC. For example, if the host does not transition to NDWin when E_{RD} or E_{WR} or E_T becomes zero, the NSC will autonomously switch to NDWin mode and start with the management activities. Therefore any uncompleted or future IOs issued by the host will be processed in the NDWin mode and may suffer from long latencies. To avoid this, the host needs to be somewhat conservative and move the NVM-set to NDWin a bit ahead of the expected DTWin expiry.

3.2.3 De ciencies of latest NVMe PLM

In a single host environment, a host module can consult the NVM-set log page to co-ordinate with NSC. The module can also allocate DC to different applications on the host, and this functionality can remain unchanged even in multi-host environment. However, multi-host environment requires a separate PLM coordinator(PLMC) to coordinate access to a shared NVM-set from multiple hosts. The storage server forms an ideal place for PLMC to live. The PLMC must support multiple QoS classes based on the limits on the tail latency. All QoS classes will have the same underlying device access latency in the absence of management operations; therefore, the difference will be mostly in tail latency. Note that the latency of concern here is simply the device access latency. The end-to-end latency will consist of at least four components: (a) host-side IO dispatching and IO com-pletion latency, (b) network transit latency, (c) NVMe queuing and queue handling latency, and (d) device access latency. QoS classes' de nition is most meaningful in terms of the

end-to-end latency, and thus control over the overall tail latency needs to consider all these four components. Therefore would require decomposing the end-to-end tail latency into its constituent elements and managing each part suitably. However, this chapter is only focused on (d).

3.3 Deterministic Service with Shared NVM-sets

3.3.1 Proposed PLM Coordinator

Figure 3.2: Proposed Coordinator for DTWin Allocation

The proposed PLMC is shown in Figure 3.2. It supports multiple application classes based on the tail latency requirement. The PLMC is intended to be lightweight and can gracefully handle the uncertainties in IO accesses and the NDWin state's transition. We also consider several policies and show their comparative performance on several real-world storage traces.

The con guration is shown in Figure 3.2. For scalability reasons, the PLMC is not involved in the data path and does not directly monitor IOs. Instead, the host is trusted to convey their usage to the PLM Coordinator accurately and abide by the limits determined by it. On its side, the PLMC should learn the needs of the workloads run by each host and supply the DC accordingly. The protocol should also preserve and use the existing interfaces as much as possible.

We assume that the remote NVMe device access is done via the NVMe-oF (NVMe over fabric) protocol [Minturn and Metz (2015); Guz et al. (2017a)]. This protocol is a simple extension of NVMe protocol wherein the host prepares a “capsule” with all the necessary command parameters and sends it as a message over the network to be deposited into the NVMe command queues residing on the target side. The completion queue is normally located on the host side, wherein the target deposits the completion capsule.

In the original (single-host) DTWin mechanism, a host reads the PLM log page that includes the parameters listed in the Table 3.1. The PLM log page per NVM-set is updated and managed by the NSC as a part of the IO completion status. Thus, by making the PLMC look like another host, it too can directly read the log information. (Ideally, the PLMC should interface with the NSCs to have more detailed access to NSC's IO management as shown in Figure 3.2, but we do not consider this approach in our initial implementation.) The PLM also needs a separate message interface with each host to handle DC requests and provide allocations. The NVM-set pool is shown in a rectangle, and individual NSC is associated with each pool. The NVMe-sets shown in green are the DT-active NVM-set, whereas the grey-colored NVM-set is in NDWIN. Both green and grey NVM-set contains the same copy of data. The solid bidirectional line represents hosts issuing IO requests to devices. The dotted black bidirectional line represents hosts' communication with the PLM controller for DTWin attribute allocation.

To obtain deterministic latency, a host needs to access the device during DTWin respecting DTWin attribute values. We assume that each host requests allocation of DC from PLMC at the start of DTWin. If the host receives fewer than the required counts, it may continue sending its IO requests to the NSC, and there may cause force transition to NDWin by NSC. The number of write requests appearing during the deterministic window determine the duration of the NDWin period.

3.3.2 Control Architecture of PLM Coordinator

The PLMC's primary role is to estimate or receive the DT count needs of each of the participating hosts and make the allocation. As stated earlier, PLMC residing locally at the storage server and can register itself with the device as another NVMe host. It is also possible to locate the PLMC at the host side, acting as one of the requesting hosts for the shared NVMe set. For fault-tolerance purposes, the PLM host can be dynamic using standard leader election algorithms [Gharehchopogh and Arjang (2014)], which we do not focus upon in this chapter.

The key parameter required by PLMC is the an estimate of the DTWin count (DC) allocated to each host for the next DT window. We have found the storage traf c to be generally quite irregular and nonstationary. Since the well-known time-series prediction methods scuh as Kalman Filter, ARMA models, etc. assume at least quasi-stationarity, they were not found to be useful, and we used a simple exponential smoothing based prediction. We will comment on this aspect in section 3.5.

Let $R_j^{pmq}(n)$ denote the measured request count (traf c) at the n-th scheduling period for the j-th class and $\hat{R}_j^{ppq}(n)$ their smoothed estimated in the same period, with $0 < \alpha < 1$ as the smoothing constant. Then,

$$\hat{R}_j^{ppq}(n+1) = \alpha R_j^{pmq}(n) + (1-\alpha)\hat{R}_j^{ppq}(n) \tag{3.1}$$

The DC allocations are are done before the scheduling window starts and do not change during the window. We assume that each host speci es some minimum DC requirement; and any excess remaining device DC's are split up among the hosts based on some policies, two of which are discussed in the following.

Let $D_j(n)$ be the DCs for the j-th device, which is as same as C_{ARD} of the j-th device during n-th DTWin. Let $DT_i^j(n)$ and $DT_j^{min}(n)$ be the total and minimum DCs allotted respectively for the j-th host and the i-th device at the n-th window, where j =1,2,...,k.

Obviously,

$$D_i(t) = \sum_{j=1}^{k} DT_i^j(t) \tag{3.2}$$

3.4 DT Count Allocations

As stated above, PLMC uses a simple exponential smoothing based estimation of the DTWin count needs of various classes, which we call *Coordinator Initiated Prediction*. It is also possible to do *host based prediction* and convey those to PLMC. The hosts may have better estimates of the traffic and thus their estimation may be preferred, however, the quality of estimate may vary. For our trace based experiments, we can count exactly how many requests will happen in the next window and supply that to PLMC. Thus the sole purpose of host based prediction for us is to simply use it as a baseline to determine how much better the allocation policy can do if we knew the DC requirements exactly in every DTWin period.

We consider the following two policies for allocating the available DC among different classes:

- Policy I (Strict Priority): Here we first estimate the minimum average DC for each host, say M_i for the i-th host. Let $M = \sum_i M_i$ denote the total minimum DC for all k hosts and D the total count for the device, which is as same as C_{RAID}. If $M \le D$, then each host can be assigned M_i DCs, for $i = 1, 2, \ldots, k$. If $M > D$, we use bin packing, assuring that each host will be allocated at least their minimum DC required. Therefore, the allocation starts with the estimated count for the high priority until we run out of counts.
- Policy II (Fixed Ratio): The minimum average DC is computed as Policy I and can have two different cases. In case the sum of the minimum DC less than D, we deal similarly as earlier. In the other case, each of the hosts is assigned the minimum required DC. The remaining DCs are allocated to each of the hosts based on the QoS-based ratio. For three QoS classes, High, medium, and low, the medium is assigned double as much as low QoS, and the high is given double as much as medium.

Table 3.2: Notation

Symbol	Explanation
$W(n)$	Actual duration of nth scheduling Window.
$W_d(n)$	Actual duration of nth deterministic Window, which can be less than or equal to T_{DW} at nth window
$W_{nd}(n)$	Actual duration of nth non-deterministic Window such that $T_{NDWL} \leq W_{nd}(n) \leq T_{NDWL}$
C	Available DT Count for the device, which is as same as C_{RD}
$C^{res}(n)$	Residual DCs after minimum allocation to all classes during $W_d(n)$
C_j^{min}	Required minimum DC for j-th class
$C_j^{used}(n)$	Consumed DC for j-th class during $W_d(n)$
$C_j^{alloc}(n)$	Allocated DC for j-th class during $W_d(n)$
D_j^{min}	Minimum required allocation for class j
$_j(n)$	Allocation of excess DC to class j (under strict priority or fixed ratio policies)
$L_j^{(n-1)}(n)$	Tail Latency measured for j-th class during $W(n)$
L_d	Latency of 4KB IO in a DT period
L_{nd}	Latency of 4KB IO in a ND period
r_j	Fixed ratio for j-th class to assign the residue DC
$R_j^{(n-1)}(n)$	Measured Traffic (#IOs/DT-period) for j-th class during $W(n)$
$R_j^{(n)}(n)$	Predicted Traffic (#IOs/DT-period) for j-th class during $W(n)$
T_{DW}	Maximum Deterministic Window Timeperiod
$u_j(n)$	Allocated DC utilization for j-th class during $W(n)$
$v_j(n)$	Deficiency of DC during $W(n)$

3.4.1 Analysis of DC Allocations

Consider k QoS classes for applications, denoted as $Q_1, Q_2, \ldots, Q_k$, with tail latency requirement of L_i for class Q_i. Also consider p identical NVM-sets $S_1, S_2, \ldots, S_p$ in a NVM-set pool. In the following we use the index n to denote the n-th window DT/ND window. The n-th DTWin is of duration $W_d(n)$, followed by a NDWin of duration $W_{nd}(n)$. We define the total duration of these as the scheduling period and denote it as $W(n)$. Obviously,

$$W(n) = W_d(n) + W_{nd}(n), \text{ for } n = 0, 1, \ldots \tag{3.3}$$

The complete set of notations are given in Table 3.2. The maximum duration $W_d(n)$ is $T_{DW}(n)$ – the specified period (100ms or 400ms in our experiments); however, if all of the

available DCs are exhausted early, $W_d(m,q) < T_{DW}$, the window also ends prematurely. The premature end of the window would include the service of all the requests arriving during the window. The duration of the non-deterministic period depends on the write request behavior during the preceding DTWin.

In each scheduling period, we need to distribute the total DCs (C, as same as C_{RD}), among the k classes (applications) such that the corresponding target latency requirement is met and no counts are wasted (i.e., reserved for a class that does not use them). In particular, the allocation of DCs should consider the following three aspects:

Definition 1 (Utilization). The utilization of a host is defined by the ratio between the number of DCs consumed vs. allocated. The utilization of j-th class during $W(m,q)$ is given by:

$$u_j(m,q) = \frac{C_j^{used}(m,q)}{C_j^{alloc}(m,q)} \tag{3.4}$$

Definition 2 (Deficiency). The deficiency of a host is defined as the fraction of requests that are not covered by the allocated DC. The deficiency of j-th class during $W(m,q)$ is given by:

$$v_j(m,q) = \begin{cases} \frac{R_j^m(m,q) - C_j^{used}(m,q)}{R_j^m(m,q)} & R_j^m(m,q) > C_j^{used}(m,q) \\ 0 & \text{Otherwise} \end{cases} \tag{3.5}$$

Definition 3 (Tail Latency) The 90% tail latency is measured for each of the considered QoS classes. The latency to execute to each of these requests could be deterministic L_d or non-deterministic L_{nd}. Therefore, the tail latency, $L_j^{(m,q)}$, of the j-th class during $W(m,q)$ is defined as

$$L_j^{(m,q)}(m,q) = \begin{cases} L_d & \text{for } R_j^{(m,q)}(m,q) \le C_j^{alloc}(m,q) \\ L_{nd} & \text{Otherwise} \end{cases} \tag{3.6}$$

where $C_j^{alloc}(m,q)$ denote the DC allocated for the jth class during $W_d(m,q)$.

The $C_j^{alloc}(m,q)$ can be expressed as follows. If the total predicted traffic $\sum_{j=1}^{k} R_j^{pq}(m,q)$ is below C, then we allocate the predicted value directly, i.e., $C_j^{alloc}(m,q) = R_j^{pq}(m,q)$. Otherwise

all the QoS classes will get the required minimum DC, DC_j^{min}. The residual DC $C^{res}(n,q)$ (i.e., the counts remaining after the minimum allocation) can be distributed among the classes based on their QoS class priority; this additional allocation corresponding to jth class denoted as $\alpha_j(n,q)$, i.e.

$$C_j^{alloc}(n,q) = \begin{cases} R_j^{pred}(n,q) & \text{for } \sum_{j=1}^{q} R_j^{pred}(n,q) \le C \\ D_j^{min} + \alpha_j(n,q) & \text{Otherwise} \end{cases} \tag{3.7}$$

To calculate $\alpha_j(n,q)$, we use the following two methods for assigning $C^{res}(n,q)$ to various QoS classes:

1. Strict Priority: The highest QoS class is assigned the DC required to match the predicted traf c for the current window, if $C^{res}(n,q)$ is enough. Otherwise, $C^{res}(n,q)$ is allocated to the next lower class, until we run out of the counts. (Obviously, the last class to get an allocation may get less than its full requirement).

2. Fixed Ratio: Here $C^{res}(n,q)$ is split among various classes based on a prede ned set of ratios r_j corresponding to different classes.

3.5 Evaluation of the Proposed PLM

Since the PLM feature is relatively new, currently, there are no commercially available SSDs that support it. Consequently, the only direct way to evaluate PLM capability is to use a comprehensive simulation model of SSDs supporting PLM. For this, we built the evaluation capabilities around the existing MQSim SSD simulation package as described below.

MQSim [Tavakkol et al. (2018)] attempts to build a very detailed and realistic model of SSDs. It explicitly represents the ash device characteristics and operation, the SSD's internal architecture, detailed FTL operations, device-level caching, and host interfacing. It provides an exact representation of both SATA and NVMe interfaces, of which we use NVMe differentiated queuing feature to support different QoS classes.

MQSim simulates nearly all aspects of SSD structure (chips, dies, planes, blocks, pages), buses (channels and inter-chip/die buses), and most non deterministic background operations associated with nonvolatile media (logical to physical address mapping, page reads, out-of-place page updates, consolidation of pages, garbage collection, block erasure, wear leveling, etc.). The developers have validated its performance against several real SSDs and found them to be very close, with the response time errors of 11% (average) and 18% maximum.

However, MQSim has some limitations and does not contain all the features necessary for building the PLM. It does not support the concept of NVM-sets or other newly proposed NVMe v1.4 features. Consequently, we embarked upon an extensive effort to understand the implementation and enhance it for our needs, as explained below.

3.5.1 Enhancing MQSim to Support PLMC

In the absence of NVM-sets, PLM emulation would require multiple SSDs, but MQSim simulates only a single SSD. It also does not support multiple hosts. For the latter, we assumed that each IO ow de ned in MQSim is a host for our purposes. We emulated multiple devices by virtualizing the entire address space into logical address ranges pertaining to the the number of required devices. MQSim also assumes that the logical address space is divided amongst concurrently running IO ows (i.e each ow accesses its own address range). We changed this aspect so that the requests from all ows are served from the same address range.

Proceeding to the PLM features, we rst had to introduce the notion of IO Determinism by stalling maintenance activities and carry them out at a later time (during NDWin period). MQSim has two triggers for garbage collection (GC): (a) soft trigger, that periodically looks for blocks with very few valid pages and maps those pages out, thereby enabling the block for erasure, and (b) hard trigger, which is initiated when the number of clean blocks falls below some limit. The soft-trigger GC can be postponed without any ill effects, but the

hard-trigger cannot. We switched off the soft trigger GC so that no GC takes place during the normal (and supposedly DTWin) period and is only triggered during the NDWin period on command. The hard trigger GC remains untouched as it is intended for the NDWin period.

For the DTWin, NDWin and DC features, we introduced some parameters that can set at the start of a simulation run. These include (a) Maximum DTWin and NDWin period (in nanoseconds), after which the SSD will autonomously transition to the other state, and (b) DC value, from which we allocate counts to all the hosts. MQSim also does not have a provision for measuring the tail latency. We have included reporting 90th percentile tail latency experienced by the host when the DTWin period ends or exhausts all allocated DCs during the given window).

Simulating NDWin period with actual maintenance operations (e.g., garbage collection, wear leveling, block consolidation, erasure) is dif cult in MQSim; therefore, we emulate unpredictable delays during NDWin period by introducing additional latency along with the soft GC which may be triggered during the NDWin period. This delay is assumed to follow an exponential distribution.

3.5.2 Emulation of PLM on Real Server

To obtain some real experimental results on the DTWin-like mechanism using regular SSDs, we use a server with Intel's Apache Pass persistent memory (PM) modules installed (in addition to the SSDs). We use two physical SSDs so that at least one of them is always in the DTWin mode. The PM is used to persist all the writes during DTWin period of an SSD, so that it only receives reads, which in turn helps make its latency highly deterministic. During the NDWin period, these writes are ushed from PM to the SSD concurrently, along with other reads/write operations issued during the NDWin period. Since it is not possible to schedule SSD maintenance operations only during the NDWin period, we introduce an additional delay to emulate these operations during the NDWin period. Since the writing

Table 3.3: DataSet Con gurations

DataSet	High	Medium	Low
DS_1	420	0.78 High	0.71 High
DS_2	480	0.83 High	0.72 High
DS_3	165	0.85 High	0.7 High

Figure 3.3: Tail Latency in isolation vs sharing

intensity to the SSD is relatively low, we avoid the triggering of garbage collection, wear leveling, etc., during limited periods when our experiment runs on the system.

3.5.3 Workloads and Con gurations Used

For evaluation, we used the Systor 2017 trace [Repository (2020)], a month-long virtual desktop infrastructure (VDI) read-intensive trace that consists of wide variations of IO size. We combined a few selected portions of the original trace to create a variety of workloads with distinct IO intensities. Table 3.3 lists three datasets. Each of these datasets consists of three workloads with different IO intensities(reported as average 4KB requests per DTWin in Table 3.3), representing Qos classes as high, medium and low.

We evaluated several different con gurations w.r.t DC and DTWin period length of the NVM-set for each of these datasets. A large DTWin reduces interactions between the hosts

Table 3.4: Parameter Con gurations

Con g	DC	T_{DW}
1	1:2 M	100ms
2	1:4 M	100ms
3	2:0 M	400ms
4	2:5 M	400ms

and PLMC and reduces overhead, and may reduce the burstiness due to the aggregation effect. However, since DCs are readjusted only at the beginning of each window, a large window will reduce the effectiveness of the control.

Because of this trade-off, we have used DC and DTWin period (T_{DW}) con gurations as represented in Table 3.4. Here M is the average number of DC required by all three workloads and is the same as M°, stated in 3.4 (which we determine by analyzing the trace). In all cases, the available DC of an NVM-set at any DTWin period is set above the average value to handle traf c variability.

3.5.4 Evaluation Results

For evaluation, we observed how the DT count allocation is distributed amongst three classes relative to the 3 metrics considered in Definition 1,2 and 3, namely (DT count) utilization, (DT count) de ciency, and resulting 90% tail latency respectively.

We used the same set of workloads and other settings for both simulation and emulation in order to enable comparison of the results. Our validation runs show that we can indeed achieve agreement between the two with respect to utilization and de ciency. However, because of numerous differences between the simulation and emulation platforms, the tail latencies are simply not comparable. In particular, the simulation model uses an extremely detailed model of a real SSD (along with NVMe protocol latencies) from several years ago, which we did not perturb. In contrast, the emulation uses a contemporary, much lower latency, NVMe SSD. Also, because of the request tagging dif culties on the real SSD, we were unable to use different NVMe queuing priorities in the emulation. Finally,

Figure 3.4: Evaluation results for Dataset 1. The tail latency is calculated in microseconds.

the persistent memory used in the emulation has no such counterpart in the simulation. Therefore, we will not compare tail latencies across the two cases.

Figure 3.4 shows the overall set of results for dataset1. Similar results were obtained for datasets 2 and 3 and therefore not reported. The top part of the gure shows results for the emulation and the bottom part for the simulation. In each case, we have listed separate cases for strict Priority and Fixed Ratio, and under those for coordinator directed (CoD) and host directed (HoD) cases. Even though Table 3.4 shows 4 different con gurations, to avoid clutter, we report results only for con gurations 2 and 4. The right hand side of the gure reports on utilization, de ciency and tail latency measures for High, Medium, and Low QoS classes, which we discuss next. For each measure, we actually generated 10 percentile, 50 percentile (Median), and 90 percentile values, however, not all are listed in the gures for brevity. The reason to choose Median, instead of Mean, is that it is less affected by occasional large quantities, which are common due to very bursty nature.

Figure 3.5: Time series for (a)-(b) Utilization and (c)-(d) de ciency characteristics of Dataset1 with Con g 2 for strict priority and xed ratio cases. The 3 QoS classes are represented by Blue: High, Red: Medium, and Grey: Low

Utilization: In Table 3.4 we only report 10 percentile utilization. The others (50 & 90%) are not reported since they all turn out to be 100% in all the cases! A 100% utilization indicates that we are not wasting any allocated counts, a sign of scarce resources. It may be noted that even 10% utilization numbers are 1 for the HoD case. This is because in this case, we know exactly what is needed. The CoD numbers are much lower because the PLMC does not know the precise needs and therefore has to be generous in its estimation. The key point to note is an extremely close agreement across simulation and emulation.

Figures 3.5(a)-(b) show the utilization for each of QoS classes over the deterministic windows for strict priority and xed ratio respectively under CoD. We see a good agreement between the two. As seen from this gure, the excess DT count allocation resulting from

Figure 3.6: Comparison of Median tail latencies for Dataset 1.

overestimation of the traf c results in less than 100% utilization. All 3 classes experience roughly the same level of utilization, with the low QoS class dominating slightly.

De ciency: For de ciency, we ignore the 10% values but show 50% and 90% in Table 3.4. It is seen that the median de ciencies are rather small: for HoD, we expect them to be small, but for CoD, they are small since the available counts are signi cantly higher than the average. Overall, there is a very good agreement between emulation and simulation. As expected, the 90% numbers are much more variable, and the agreement between emulation and simulation is somewhat worse, but still quite good overall. It is worth noting that both emulation and simulation data is based on about 9000 DT Windows for con guration 2 (and 2250 for con guration 4), which means that 90 percentile values are expected to be rather variable. Figures 3.5(c)-(d) represent the de ciency for each of QoS class over the deterministic windows for strict priority and xed ratio respectively under CoD. The gure shows an opposite trend of utilization. The agreement between simulation and emulation is also clearly demonstrated.

Tail Latency: For latency as well, we report 50% and 90% values. As stated before, the latencies are not comparable across emulation and simulation, and we shall make no attempt to do so. Also, since the simulation model is far more granular here, we henceforth

Figure 3.7: Comparison of Median tail latencies for Dataset 4.

comment only on simulation results. The median values are shown more clearly in Figure 3.6. It is seen that there is very good agreement between HoD and CoD values across all three QoS classes and for both strict priority and xed ratio cases. This is remarkable in view of extremely simple exponential smoothing based prediction and highly bursty nature of the traf c. It is also worth noticing that the latencies of High and Medium classes are controlled quite well at the cost of Low QoS class.

Looking at the 90% tail latency, we again nd good control over the latency of High and Medium QoS classes, with a signi cantly lower latency for the High QoS than Medium QoS. However, when we look at the 90% tail latency values, the agreement becomes poorer. For example, for con g 2 and strict priority, the CoD values are (548, 773, 6589) whereas the HoD values are (417, 924, 7155). The tail latency is most important for High QoS, where the CoD is about 31% higher. Such an accuracy is actually quite good, again, considering the stress case considered here: very high burstiness and synchronized traf c. The direction of error reverses for the Medium priority, which is entirely expected from the conservation law [Kant (1992)]: A poorer treatment of high priority will result in better treatment of the lower priorities. Finally, there is a decent agreement on the Low priority, but latency is not important for that class. One area of concern that emerges from our eval-

uation is estimation of DT count needs by the PLMC, as discussed in section 3.3.2. Since time-series methods are unlikely to be useful, a machine learning based traf c predictor is another possible approach. However, even a complex neural net is unlikely to work well unless the traf c does have signi cant repeating patterns.

Given this, we believe that a hinting mechanism by the hosts (e.g., by the applications running on the hosts) is the only practical method. However, the details of such an approach are beyond the scope of this chapter.

We next simulate a less challenging situation where we create another dataset (henceforth called Dataset4) from Dataset1 by adding an offset of 200 ms and 400 ms to Medium and Low priority traf c respectively. This change desynchronizes the peaks and results in a traf c pattern where satisfying the requirements of all classes becomes much easier. Figure 3.7 shows the variation of tail latency for Dataset4. It is seen that the latency drops drastically, by almost 30X, as compared to Figure 3.6. With such low latencies, the QoS issue is moot, and all classes get essentially the same treatment. As to the agreement between HoD and CoD, it is excellent for xed ratio, and very good for strict priority (16% error for high prioriy). However, any discrepancy in this case is irrelevant, since all QoS objectives can be more than met.

3.6 PLMLight : Emulating Predictable Latency Mode in Real SSDs

As discussed in this chapter till now, PLMC considers only read requests and distributes $read_D C$ among hosts. This is because write intensive workloads pose a more challenging scenario. For consistency issues, one cannot delay writes and need them to be performed as and when requested. Additionally, consuming $write_D C$ can force a device to transition from DTWin to NDWin. This in turn can cause high latency values. We hence propose a modi ed version of PLMC - PLMLight, which caters to both read and write requests in regular SSDs while preserving the details of the PLM feature. Thus helping in achieving low latency for High class applications.

Figure 3.8: An Illustration of PLMLight Architecture

PLMLight is comprised of two components - PLMLC (or PLMLight Coordinator) and a WHA (Write Handling Agent) at every server. As shown in Figure 3.8, the former resides in the control path outside the server and acts as a host to communicate with the storage server. It can keep a check on the amount of traf c pushed by each host via message passing. This helps in allocating $read_D C$ for the hosts. In order to control non-deterministic latencies triggered by writes, we cap the $write_D C$ usage on the SSD and direct excess writes to a different device (ex. Intel Optane SSD). This redirection is carried out by the WHA, which (as shown in 3.8) resides in the storage server along with NSC. This helps it in directing writes to the appropriate write IO location.

Figure 3.8 shows the entire architecture of PLMLight where PLMLC allocates $read_D C$ to each host accessing the target. Each target is comprised of multiple NVM subsystem(s), i.e SSD(s), and Intel Optane as the device used for handling excess writes. Additionally, the PLMLC maintains a per-device consolidated $write_D C$ to make sure that on exhaustion of $write_D C$, the device seamlessly transitions to NDWin. The NVMe-oF (NVMe over Fabric) [Minturn and Metz (2015); Guz et al. (2017a)] protocol is used by the hosts to communicate with the PLMLC. This helps them to access the NVM set's log-page so as to gauge $read_D C$ information. The NSCs in the NVM subsystem control the NVM sets. We

assume that the hosts are well behaving, i.e. they reliably respect and convey the usage of allocated counts to the PLMLC. PLMLC uses similar workload estimation techniques (i.e. exponential smoothing) as PLMC to estimate required DC. It allocates on the basis of this estimation and depending on the host's QoS class. The WHA is removed completely from any communication with the host and independently decides the location of the write IO on the basis of the device characteristics.

As mentioned, we distribute $read_D C$ depending on the workload characteristics and the QoS class of the application. On this note, we can treat the Low class poorly by making it wait and perform its reads during an NDWin period. However, for $write_D C$, we cannot apply the same treatment. This is because stalling write operations would cause consistency issues. Hence writes need to be completed as soon as they come in. Hence we refrain from refraining between QoS classes for writes and try to ensure that all writes pertaining to all QoS classes receive the same treatment while the reads receive QoS differentiation.

3.7 Evaluation of PLMLight

For our evaluation of PLMLight we utilized the widely used and accepted FIO Benchmark. We use FIO to create two workloads with differing characteristics to stress test PLMLight. The workloads were executed as three different processed on three different CPUs, thus capturing the notion of three hosts. The workload details are as follows:

1. Workload I: This workload is a write intensive workload with a read write mix of 30:70. The request size is kept constant as 4KB to observe PLMLight's reaction towards single block requests.

2. Workload II: This workload is read intensive with a read write ratio of 70:30. The request sizes range from 4KB to 128KB. This helps in testing PLMLight's performance in the presence of variable and larger requests.

Due to the use of widespread use of caching and prefetching in computer systems, we used 100% random workloads. We can observe this random nature of both workloads in Figure 3.9a and Figure 3.10a, which are 2-second snapshots of Workload I and II respectively. For our evaluations we use the 90%-ile, 92%-ile, 95th%-ile, 97th%-ile, 99%-ile and 99.9%-ile latency values experienced by the applications for both reads and writes.

(a) Workload Snapshot (b) Read Tail Latency (c) Write Tail Latency

Figure 3.9: Evaluation of Workload-I

In our experiments we performed a control (or baseline) experiment with three workloads pertaining to three hosts without any QoS differentiation. Our graphs show a single case for all three hosts. This is because in the absence of QoS treatment, all three hosts receive similar treatment with negligible differences. The DC allocation was carried according to xed ratio policy. The available read_DC and write_DC for every DTWin are the same and considered as 80% of the average read and write IOs per DTWin. The DTwin period is considered as 1 millisecond. Therefore, 80% of the read and write IOs will be served in 1 millisecond. PLMLight does not distribute write_DC among the QoS classes. All QoS classes share the write_DC on a rst-come- rst-serve basis.

For write intensive workloads (ex. Workload I), the read latency can extend to a few milliseconds, as shown in Figure 3.9. The baseline extends to millisecond latency beyond the 90%-ile. As noted before, the request sizes are small, i.e. 4KB. Figure 3.9b and Figure 3.9c show that PLMLight outperforms the High QoS class in both cases. Due to the same treatment provided to writes of all QoS classes, we observe that all three classes exhibit the

same latency behavior, that too at microseconds level, which is a de nitive improvement on the baseline case. This is because most writes are handled by the Optane and are ushed to the SSD during the NDWin period. For the read latency we observe that High QoS performs better than the baseline but Medium and Low suffer. This is expected as both Medium and Low are treated poorly in comparison to High. However, the QoS differentiation is preserved.

(a) Workload Snapshot (b) Read Tail Latency (c) Write Tail Latency

Figure 3.10: Evaluation of Workload-II

In Workload II, the IO size varies with a small fraction of writes (writes with less than 64KB IO size) being persisted in the Optane, while the rest go to the SSD. Figure 3.10b (i.e. the read tail latency comparison) shows how the High QoS class outperforms the baseline read tail latency by 80% for 99%-tile and 55% for 99.9%-tile, which is almost similar to the performance achieved with the read-intensive small IO workload (Workload I). The read latency of Low and Medium QoS classes noticeably improves from its Workload I counterparts by avoiding write IO execution in SSD. Although, a small amount of write is present in the workload, persisting a portion of it in Optane improves the write tail latency signi cantly in PlMLight as compared to the baseline (shown in Figure 3.10c).

3.8 Conclusions

This chapter examines the policies for allocating DTWin counts for NVMe v1.4 predictable latency model (PLM) capability in a shared environment. We propose a PLM

coordinator (PLMC) design that can work without any changes to the current NVMe v1.4. We evaluate the proposed mechanisms in two ways: by making changes to a comprehensive SSD simulator (MQSim) and by using real implementation on a server with persistent memory to achieve comparable DTWin periods. The experimental evaluation of the mechanism demonstrates that it can substantially reduce tail latency and help deliver predictable latency to the high QoS classes. However, the PLMC could bene t signi cantly from improved traf c prediction, which could perhaps be achieved via “hints” from the applications. We also propose another version of PLMC called PLMLight which handles both read and write execution in a real emulated environment due to the PLM feature still not being available in modern day SSDs. We evaluate it too and show how we can segregate writes to a different device while preserving the QoS differentiation in read requests.

CHAPTER 4

CONFIGURING AND COORDINATING END-TO-END QOS FOR EMERGING STORAGE INFRASTRUCTURE

4.1 Overview

Modern data center storage systems are invariably networked to allow for consolidation and exible management of storage. They also include high performance storage devices based on ash or other emerging technologies, generally accessed through low-latency and high throughput protocols such as NVMe (or its derivatives) carried over the network. With the increasing complexity and data-centric nature of the applications, properly con guring the quality of service (QoS) for the storage path has become crucial for ensuring the desired application performance. Such QoS is substantially in uenced by the QoS in the network path, in the access protocol, and in the storage device. In this chapter, we de ne a new transport level QoS mechanism for the network segment and demonstrate how it can augment and coordinate with the access level QoS mechanism de ned for NVMe, and a similar QoS mechanism con gured 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 (PM) devices located along with the storage but requiring much lower latency than storage. We demonstrate that a proper coordinated con guration of the 3 QoS'es on the path is essential to achieve the desired differentiation depending on where the bottlenecks appear.

In this chapter we consider a scenario of 3 classes of Normal Priority Applications (termed as High, Medium and Low) and a single class of High Priority Applications (i.e. Urgent) as shown in Figure 4.1. A section of this work was earlier published[Joyanta Biswas (2021)] in collaboration with Dr. Joyanta Roy.

Figure 4.1: QoS Classi cation

4.2 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 evolution in storage technologies has made it possible to continuously improve the size, speed, cost, and exibility 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 ll 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., 4KB 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.

(a) (b)

Figure 4.2: (a)Overview of end-to-end data center organization and (b) comparison between local and end-to-end storage access

4.2.1 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 it 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 4.2a. The emerging persistent memory (PM), that provides nonvolatile storage with latencies of about an order of magnitude slower than DRAM (but much faster than storage) can also bene t 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 signi cant part of the overall access latency, which was not an issue for the older HDD-based storage. The oversized impact of net-

work latency is also facilitated by highly efficient storage access protocols such as the Non-Volatile Memory Express (NVMe) protocol Cobb and Huffman (2012). As shown in Figure 4.2b, when accessing an SSD locally, the access latency is of the range of 30-100 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/sec (or 32 Gb/sec), few SSDs can easily congest even a 100 Gb/sec 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.

4.2.2 End to End QoS Differentiation

With increasing demands of a 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. Furthermore, enforcing specific SLA requirements involves a lot of factors including how much computing resources are allocated to the application and the storage related aspects (i.e., HDD vs. SSD, host or target side caching, etc.). Thus, we consider QoS differentiation only in terms of relative treatment of different classes of applications. We will assume that the applications have already been classified into a small number of classes (up to 4), and the QoS differentiation concerns their relative treatment in case of congestion or resource contention. Further, we consider applications to be classified into two types - throughput sensitive and latency sensitive. In the context of networked storage access, this means that we specify this relative treatment in terms of the four key components on the path: host-side, network, storage protocol, and the device. For each, a convenient way to specify the relative treatment is in terms of resource (e.g., network or storage throughput), and this is

generally the approach we will take. The SLA for the latency sensitive application class often speci es some bounds on the tail latency (e.g., 99 percentile latency), but it becomes very dif cult to control latencies directly in the face of congestion. The only exception is the class that is given priority over all others and its offered rate remains low enough that it can always be accommodated at the cost of others. Beyond this, while it is possible to speak of relative latencies (much like relative throughputs), it may not provide a meaningful treatment under severe congestion. Note, however, the relative throughput management does impact the latency, we shall discuss corresponding results.

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.

4.2.3 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 traf c. 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 traf c and show how we can protect the QRDMA PM traf c from QTCP traf c 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 rst detailed examination of mixed PM/storage network traf c.

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 dif culties associated 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 rst such exploration and it provides important insights into how E2E QoS should be con gured for networked storage.

The rest of the chapter is organized as follows. Section 4.3 discusses the background and motivation of our work coupled with the limitations of existing works while section 4.4 discusses our proposed mechanisms. We model our proposed technique in section 4.4.1, followed by extensive evaluation in section 4.5. We additionally discuss some more works of interest and their limitations in section 4.6. The chapter is concluded in Section 4.7. Table 4.1 includes some of the key abbreviations used in this chapter.

Table 4.1: Common Abbreviations

SNIA	Storage Networking Industry Assoc.	HDD/SSD	Hard Disk/Solid State Drive
FTL	Flash Translation Layer	SLC/MLC	Single/Multi (2) Level 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 Noti ca-tion
CNP	Congestion Noti cation Packet	HOL	Head of the Line
E2E	End-to-End	QoS	Quality of Service

4.3 Background and Motivation

In this section, we discuss the necessary background relating to the host-to-device storage access path. There are 4 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 chapter about mechanisms that we have either invented or harnessed to con gure a consistent end-to-end QoS treatment of storage request of various sorts.

4.3.1 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 le-system that the host has de ned over the storage space allocated for it. The le system maps the IO requests to the allocated storage, which is viewed simply as a sequence of “logical blocks”. These are typically 4KB 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 ef ciency, 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 16KB 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 in order 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 end-to-end treatment of IO requests that result in a device access. In fact, caching is so pervasive that it may occur at multiple points in the storage path. For example, most devices do have a device cache, and the storage server itself often has a shared cache.

4.3.2 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 [NVMe (2019)] has recently become dominant and uses several mechanisms to achieve high performance including a direct, hardware-supported interface to the CPU, distinct sets of “queue pairs” (request submission and completion queues) tied to individual CPU cores, largely user space IO handling, interrupt moderation or polling to handle completions, etc. (See Figure 4.3a taken from [OSR (2014)]).

NVMe is a rather complex protocol and supports many features relevant to SSDs including a differentiated queuing structure shown in Figure 4.3b. There are 5 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 3 are for throughput-centric

traf c. 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 target without changing the basic access semantics. NVMe over Fabric (NVMe-oF) [NVMeOF (2019)] provides this capability by encapsulating the submission and completion queue entries into transport-independent “capsules” for transfer between the host and the target. 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.

(a) (b)

Figure 4.3: QoS Aware Queuing in NVMe

NVMe-oF is intended to run on top of an end-to-end network transport so that the storage server can be located anywhere on the data center network and still be accessible to all

Figure 4.4: Internal Architecture of an SSD

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 [Bensley et al. (2017)] [Voloshin (2022)] protocols that we shall use as baselines. One is the so-called Data Center TCP (DCTCP) [Alizadeh et al. (2010a)] and the other is Data Center Quantized Congestion Noti cation (DCQCN) [Zhu et al. (2015)]. The latter builds a lossless RDMA service on top of RoCEv2 by using some data center Ethernet features as described later.

4.3.3 Characteristics of Emerging Storage Technologies

The dominant storage devices in modern systems are SSDs which are based on the “ ash” technology. The key characteristic of this technology is that it does not allow overwrites. Thus, a “cell” of a ash (which can store 1, 2, 3 or 4 bits depending on technology) must rst 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 target value. 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 4-16KB, and all reads/writes occur at this granularity. Figure 4.4 shows backend details of the SSD where the aforementioned pages are further grouped into “blocks” (or ash-blocks, different from storage system notion of a block), of sizes 128KB-8MB. Erasures must occur at the level of ash-blocks, which introduces further complexity. The ash-blocks are further grouped into “planes”, and a erasure generally blocks the entire plane. Multiple planes form a chip-die, and a physical ash-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 rmware of the SSD known as the Flash Translation Layer (FTL). FTL is generally vendor-speci c and proprietary, although open-source versions such as OpenSSD [OpenSSD (2022)] do exist. Figure 4.4.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 (ex., the NVMe submission queues explained in 4.3.2) 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” [Rho et al. (2018)] [Kim et al. (2018)] [Kim et al. (2019)], wherein the host provides hints

to the FTL about the life-time of the ash-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 [Alazzaw et al. (2020)], 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 s [Zheng et al. (2019)]) and also as PM with substantially lower access latencies, typically less than 1s. On the flash side, even the rather inexpensive SSDs available today are capable of driving up to 25-35 Gb/sec IO throughput [Davis (2016)] with nominal latencies of under 100 s [Tallis (2019)].

With storage semantics, the PM access sizes will typically be large (e.g., 4KB), 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 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 episodes. Furthermore, the switches (in addition to the endpoints) should also provide preference to the PM traffic. We will show

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

that such network level intervention can reduce the PM latency substantially.[2]

4.3.4 Network Issues for Remote Storage Access

As stated earlier, the high throughput of modern storage devices can easily congest even high-speed 100 Gb/sec 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 [Bensley et al. (2017)] and available in some switch and NIC implementations [Voloshin (2022)].

Both DCTCP and DCQCN depend on the noti cation of congestion to the sending endpoint via standard mechanisms. In case of DCTCP, it is explicit congestion noti cation (ECN) and involves 2 bits in the packet header. One is set when any switch on the forward path experiences congestion, de ned as the queue length in the switch exceeding some prede ned 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 noti cation packet (CNP) to the sender. DCQCN also utilizes the data center Ethernet QoS feature called PFC (priority ow control) for selectively pausing the ow towards a congested destination. The standard PFC mechanism operates at layer 2 and pauses the ows 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

[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.

based protocol is to provide low latency for the small transfers needed for remote PM access. However, when PM and storage traf c share the same network, it is also important to give preferred treatment to PM traf c 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 traf c.

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 de ned for wide area networks, with packet loss-centric QoS treatment. Furthermore, DSCP de nes a “hop-by-hop” QoS treatment (in each router), rather than a uniform end-to-end treatment. Thus DSCP, as de ned, is not useful for QoS differentiation. We later on also discuss other QoS aware solutions (ex., HOMA, PDQ,^{2}DCT, D^{2}TCP and D3) along with their limitations in 4.6.2.

4.4 E2E QoS Differentiation in Networked Storage Transfers

In this section we divide our E2E QoS discussion into three major parts. We rst talk about our proposed QTCP and QRDMA modi cations in the network along with meeting QoS requirements for ultra low latency traf c. Following this we discuss how we achieve QoS differentiation in the storage end. Finally, we explore appropriate mechanisms to mark the network packets for E2E con guration of QoS.

4.4.1 QoS in the Network

QoS aware DCTCP and DCQCN (QTCP and QRDMA)

We rst de ne how the QoS metrics are utilized by our proposed QTCP and QRDMA mechanisms in the network during a congestion episode. For both throughput sensitive and latency applications (also termed as “ ows”), we make use of their QoS objectives to

Figure 4.5: Flowchart for Quality Factor (QF) based ow control

provide QoS differentiation via relative treatment. The QoS objectives are used to calculate a ratio and we consider that all applications (classi ed into number of QoS classes) stick to their QoS throughput their lifetime. In both cases we assume the bottleneck bandwidth is known as it can be estimated from the techniques used in [Cardwell et al. (2016)]. For latency differentiation we consider the tail latency objective to be de ned by the same percentile value, ex. 90th percentile latency of 150 s[3], while for throughput differentiation we consider the minimum bandwidth requirement of a ow. As mentioned in 4.3.4, the ECN mechanism is used to detect congestion.

For our proposed mechanisms, we introduce a new QoS class-speci c measure called Quality Factor, denoted as Q_i for a class i. It is de ned 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 de ned as,

$$Q_i \quad {}_{it} \{ \; {}^{1}_{ia} \text{ where } {}^{1}_{ia} \quad p \; 1 \quad q \; {}_{ia} \quad {}^{1}_{ia} \; ; i \quad 1 :: K \qquad (4.1)$$

Where, ${}^{1}_{ia}$ and ${}_{it}$ denote, respectively, the actual smoothed bandwidth and the target bandwidth, while is the smoothing factor. Q_i is a unit-less metric as it is a ratio. A value of $Q_i \quad 1$ denotes that the ow in question has slack available, meaning that its corresponding window can be squeezed or reduced to make way for other Q_i ows. 1

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

denotes that a de cit has been detected and the ow's window needs to be increased to meet its QoS requirements. The ow rate for each ow is modulated separately as it is assumed that ows have separate connections.

We can similarly de ne Q_i for tail latency too. However, in this case, the ratio is reversed to preserve the understanding that a value of Q_i of 1 denotes slack and Q_i ¡ 1 denotes de cit as before. So in this case, Q_i looks as follows,

$$Q_i \quad L_{ia}^{1}\{L_{it} \text{ where } L_{ia}^{1} \quad p\ 1 \quad q\ L_{ia} \quad L_{ia}^{1}; i \quad 1::K \tag{4.2}$$

The congestion noti cation via real or emulated ECN mechanism reports 1-bit information per ACK (or CNP), which can be quite chaotic. Thus, the quantity of interest is the fraction of ACKs or CNPs received in an RTT window that indicates congestion. This fraction can be monitored separately for each ow 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 $_i(n)$ denotes the smoothed congestion fraction of class i at RTT n, we have:

$$_i(n) \quad p\ 1 \quad g)\ _i(n \quad 1) \quad g f_i(n \quad 1) \tag{4.3}$$

where $0 \quad g \quad 1$ 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 $f_i(n \quad 1)$ can only be 0 or 1.

DCTCP reduces the window per round-trip time (RTT) in proportion to the latest estimate of $_i$ such that in the limiting case of $_i$ = 1, the window is halved. Hence the window controlling mechanism is formulated as follows:

$$W_i(n) \quad W_i(n \quad 1)\ 1 \quad \frac{_i}{2} \tag{4.4}$$

Similar to DCTCP and DCQCN, we use the value of $_i$ (i.e., the fraction of packets that are ECN marked) along with our new metric Q_i to modulate the ow rate for the different

ows. We term this modi ed version of DCTCP as QTCP and rede ne the window size W_i for a ow i as follows,

$$W_i(n+1) = \begin{cases} W_i(n) + 1 & \text{No ECN} \\ W_i(n) + 1 - (1 - \frac{\alpha_i}{2}) & \alpha_i \geq 0 \text{ and } Q_i \geq 1 \\ W_i(n) + 1 - (1 - \frac{\alpha_i}{2})Q_i; & \alpha_i \geq 0 \text{ and } Q_i < 1 \end{cases} \tag{4.5}$$

In fact, if $Q_i = 1$, 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. QTCP refrains from using the quality factor metric when there is no congestion and even when congestion is encountered, but de cit 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 ow and so only the value of α_i is used for ow rate modulation. However, when a slack is detected, i.e., $Q_i < 1$, the QoS requirement has already been met and so the ow should be backing off so as to free resources for the other competing ows 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 ow rate in every N microsecond intervals, our proposed QRDMA also updates the ow rate in the same manner. The updated equations look as follows where RC_i is the current ow rate,

$$RC_i(n+1) = \begin{cases} RC_i(n) + 1 & \text{No ECN} \\ RC_i(n) + 1 - (1 - \frac{\alpha_i}{2}) & \alpha_i \geq 0 \text{ and } Q_i \geq 1 \\ RC_i(n) + 1 - (1 - \frac{\alpha_i}{2})Q_i; & \alpha_i \geq 0 \text{ and } Q_i < 1 \end{cases} \tag{4.6}$$

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

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 [Alizadeh et al. (2011)]. 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 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 n i.e., the current time slot and it is assumed that there are $1 \ldots I$ 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 $q_{aj}^{pi\,q}(n)$ 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,

$$e_i(n) = I\left(\sum_{j=1}^{I} q_{aj}^{pi\,q}(n-1) \geq K\right) \tag{4.7}$$

where 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 $q_{aj}^{pi\,q}(n)$ has a weak if not negligible dependence on this. Thus we can henceforth denote this event (i.e., congestion detected) as just F. 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,

$$p(n) = \begin{cases} 1 - \frac{K-1}{B(n-1)} & \text{if } B(n-1) \geq K \\ 0 & \text{if } B(n-1) < K \end{cases} \tag{4.8}$$

where $B(n-1) = \sum_{i=1}^{\infty} q_{ai}(n-1)$.

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 d_i, which encompasses all delays which 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 traf c rate is high, but may involve explicit ACKs whenever there are gaps in the backward traf c. 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 4.2 consists of signi cant terms and their descriptions used in this analysis. We now consider the two metrics of signi cance 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, $C_i = r_i / \sum_{\forall i} r_i$. 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

Table 4.2: 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		Fraction of ECN marked packets

packets $W_i(t - \overline{R_i})$ during the preceding update slot $(t - \overline{R_i})$, 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 signi cant. 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,

$$C_i(n) = \begin{cases} C \cdot r_i / \lambda_i & \text{Throughput control} \\ \dfrac{W_i(n-1)}{R_i(n-1)} & \text{Latency Control} \end{cases} \tag{4.9}$$

$$Q_i(n) = \begin{cases} \dfrac{C_i(n) \cdot R_i(n-1)}{W_i(n-1)} & \text{Throughput control} \\ \dfrac{d_i + q_{ai}(n-1)/(C_i(n-1))}{L_{it}} & \text{Latency Control} \end{cases} \tag{4.10}$$

$$\lambda_i(n) = \lambda_i(n-1) + r(p(n) - \lambda_i(n-1)) \tag{4.11}$$

Note that all units are in terms of packets and not bytes. We rst 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 $(t - q_{ai}/C_i)$.

[4] Multi-class open-system queuing formulae [Kant (1992)] can be used to gauge the range of usable values.

The value of $f(n)$ 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 $Q_i(n)$. Followed by this we update the value of Δ 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,

$$W_i(n) = \begin{cases} W_i - 1 & \text{if } q_i(n) \le \varepsilon \\ W_i(1 - \frac{\varepsilon_i(n)}{2}) & \text{if } q_i(n) > \varepsilon \text{ \& } Q_i(n) > 1 \\ W_i(1 - \frac{\varepsilon_i(n)}{2})Q_i(n) & \text{if } q_i(n) > \varepsilon \text{ \& } Q_i(n) \le 1 \end{cases} \tag{4.12}$$

$$R_i(n) = 2d_i + B\{C \tag{4.13}$$

$$q_{ai}(n) = \max(0, q_{ai} + W_i(n) - C_i R_i(n)) \tag{4.14}$$

It is difficult to seek the steady state for this system when W_i and R_i change (as seen in equation 4.12), 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 $W_i^1(n)$ can then be represented as the minimum of the estimated window size $W_i(n)$ (from our previous equations) and $U_i(n)$. Hence,

$$M = \inf_{p \in \{M+1, \ldots, G_i^{(n,q)}\}} \{q_i(R(n+1)) + M + 1\} \tag{4.15}$$

$$U_i(n) = U_i(n+1) - W_i^1(n+1) + M - 1 \tag{4.16}$$

$$W_i^1(n) = \min(W_i(n), U_i(n)) \tag{4.17}$$

$G_i^{(n,q)}$ is the time elapsed between succeeding packets $(n+1)$ and n during an RTT slot. This time measure is governed by the arrival distribution of the packet, which could be

bursty in nature.

QoS Differentiation for Mixed Traf c

We now look into the mixed network traf c that comprises of both storage traf c (with transfer sizes or 1 or more LBAs) and ultra low latency traf c (ex. PM traf c with transfer sizes of only 2-4 cachelines).

Providing very low latency to a speci c class of network traf c requires that such traf c gets the highest priority in the network switches as well, henceforth called “urgent” priority, following NVMe terminology. Since the data transfer latencies for storage traf c 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 traf c. Both types of traf c 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 end-to-end communication of priority classes.

In this work we use the urgent priority for ultra-low latency traf c in two ways: (a) Ample space reservation in the egress switch port, and (b) Giving strict (but nonpremptive) priority to the Urgent class queue over other queues. In summary, if the buffer size is B packets, the best QoS treatment for Urgent traf c can be achieved if there is a buffer reservation P for it. This traf c (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 1 packet transmission time (120ns at 100Gb/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 traf c is not throttled unless the host uses a suitable admission control mechanism to ensure that this traf c does not exceed the design limits. We utilize this prioritization technique for our evaluation of PM traf c in 4.5.4.

Figure 4.6: Extending NVMe WRR inside the SSD

4.4.2 QoS Differentiation at the Storage End

For E2E QoS differentiation, we also need to ensure that the target storage end provides QoS differentiation for incoming ows. 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, the SSD receives 2 requests of the Medium class after every 3 requests of the High class. However, how these requests are scheduled and serviced inside the actual device is a more complex problem (due to the issues mentioned in 4.3.3) and 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 4.6. We show that this simple change can be effective in gaining a tighter handle on the access latencies. Also, chip-level queues schedule requests in the size of single pages. This gives us an added level of control on the latencies. Thus we show how this throughput differentiation mechanism can also be used to provide QoS treatment for latency.

Additionally, the NVMe protocol has introduced the notion of “predictable latency mode” which allows the background operations to be con ned to certain periods known

as “nondeterministic” (NDT) periods. We have explored this extensively in [Roy et al. (2021)] to provide deterministic latency to different applications. We introduced a coordinator module called PLMC (i.e. PLM Controller) that allocates the number of “deterministic” accesses an application can perform on the basis of its QoS class. This could also be used in conjunction with our proposed NVMe WRR extension in order to provide even tighter control on the access latencies. However, that is beyond the scope of this work.

4.4.3 End-to-End Con guration 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 classi cation, 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 speci c vendors in their products. We have con gured a mechanism similar to NVMe for the SSDs as well, namely 4 priority queues including an “urgent” queue and 3 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 ne-grain since a chip might have many planes (128-1024) but would not add much value unless except in rather pathological cases of hot spots within each plane. For simplicity, we have con gured it only at the Chip level. It is possible study impact of variable granularity with this con guration by varying the number of chips and planes/chip, but that level of detail is outside the scope of this work.

The 4 QoS classes discussed so far (3 QoS classes for storage traf c and one for PM) need to be carried from host all the way to the device for each storage read/write opera-

tion in order 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 brie y discuss how an end-to-end QoS hinting can be supported in contemporary storage stacks.

With 4 QoS classes, we need to nd at least 2 bits in the headers read and write commands to convey the QoS class. NVMe read/write commands de ne a few bits in Dword (double word) #13 that relate to latency and frequency expectations of the traf c. While those bits can be leveraged for QoS, a better solution is to use two of the unused bits. These would be ignored by con gurations that do not support QoS. If NVMe is used to carry the SCSI command directly, these bits can be mapped to the “group number” eld in SCSI header, which remains unused [Mesnier et al. (2011)]. 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 suf ce for in-network QoS treatment of PM traf c mentioned above in 4.4.1. Switches cannot do deep packet inspection (DPI) to nd 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 traf c 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 traf c in the network, but one could carry the 2-bit classi cation 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 chapter.

Generally, the treatment will be derived from the application type in terms of transfer sizes, average intensity, SLA, etc.

4.5 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.

4.5.1 NeSt - A QoS Differentiating E2E Networked Storage and Memory 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 [Riley and Henderson (2010)], Omnet++ [Varga and Hornig (2008)], and MQSim [Tavakkol et al. (2018)] 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 Differentiating E2E Networked Storage and Memory Simulator - NeSt. NeSt 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 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 NeSt 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

(a) (b)

Figure 4.7: Evaluation of (a) simulation vs real environment and (b) PM access in NeSt switches.

2. Supports multiple SSDs and multiple PMs with varying device con gurations.
3. Simulates NVMe-oF with multiple transport protocol options - ex., DCTCP, QTCP, etc.
4. Contains an inbuilt traf c 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.

In order to evaluate NeSt's performance in comparison to a real E2E scenario, we compared real-life storage accesses (both remotely and locally) with their simulation equivalent con- gurations in Figure 4.7a. We used a storage workload to request data from a “Samsung 970 EVO Plus” SSD, both locally and over the network (with a 100Gbps link) using TCP as its transport protocol. We also evaluated the same experimental setup in our simulation environment. From Figure 4.7a we can observe that the MQSim SSD simulator (which

NeSt 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 40-90µs. In Figure 4.7a we also see that NeSt's E2E performance compares to the latency values exhibited in a real E2E storage access. This shows that NeSt simulates both local and E2E storage access latency accurately.

Figure 4.8: Fat tree architecture with simulated storage servers in NeSt

In Figure 4.7b we also evaluated the PM access performance in NeSt by using its inbuilt traf c generator to request memory traf c over the network. In the case of accessing PM locally as a memory device, it has been reported [Deierling (2016)] that its access latency ranges in hundreds of nanoseconds. In NeSt we observed (as shown in Figure 4.7b) that the native PM latency at the endpoint averages ¡900ns while the E2E delay for the remote memory traf c falls in the range of 3-4µs. This is an accurate representation of the performance of PM traf c, both remotely and locally as the latency values quoted in the SNIA PM Summit [Grun et al. (2018)] talk about how accessing Persistent Memory over the Fabric results in latency values ¡4µs.

For the subsequent evaluations of our proposed strict provisioning for PM traf c 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 4.8. 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 4.8,

(a) (b)

Figure 4.9: Avg. Latency comparison for Different Storage QoS Scenarios with (a) Fujitsu Lab and (b) TPCC traces

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

4.5.2 Evaluation of QoS Differentiation at the Storage End

In this section we evaluate the changes we proposed in 4.4.2 by considering three scenarios pertaining to the treatment each QoS class receives -

- Scenario 1: QoS Agnostic Protocol and Device, i.e., both NVMe and the SSD treat all ows 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 NeSt in isolation to observe the behavior

of storage workloads in the three different scenarios in Figure 4.9. We utilize the workloads published in the SNIA repository [Repository (2020)] 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 85s for Scenario 1 in Figure 4.9a. The Medium and Low classes performed signi cantly better, reporting latencies less than 70 s and 40 s 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 signi cantly 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 traf c 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 4.9a, 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 signi cant component contributing to the storage latency. As mentioned in 4.3.3, 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 4.4.2. 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 signi cant difference in performance for all 3 QoS classes as we can see in Scenario 3 for Figure 4.9a. We nally observe the desired differentiation between the three classes. The Low QoS class performs signi cantly worse to ensure that the other two classes perform better with the High QoS reporting an average latency value close to 50 s while the Medium class reports close to 60 s. It is 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 4.2.1. We also see the effect of network throughput on the storage throughput in later sections.

We also tested a less pathological case in Figure 4.9b with the three applications being a combination of the TPCC benchmark collected at Microsoft [Kavalanekar and Worthington (2007)]. The intensity of the High and Medium class traf c, 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 4.9c. 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 4.9c. Hence having QoS-aware device-level queues aids the NVMe protocol in guaranteeing storage end QoS 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 traf c entering the PM device does not need to compete with other classes of traf c, unlike storage traf c in the SSDs. We now look into the performance of our proposed QTCP and QRDMA.

4.5.3 Evaluation of QTCP and QRDMA

Analytical Model Evaluation

Figure 4.10: Comparison of our analytical modeling vs simulation

We rst compare our analytical model (described in 4.4.1) with NeSt's implementation of QTCP in Figure 4.10 using the same topology as Figure 4.8. We measure the

(a) (b) (c)

Figure 4.11: QF Convergence for Differing RTTs

throughput experienced by High, Medium and Low QoS applications, with an offered load of 90Gbps, 60Gbps and 30Gbps respectively. The bottleneck link has the standard data center 100Gps link bandwidth. Assuming the threshold value to be 140 K (K being approximately $0.17Cd$, where C is the bottleneck capacity and d is the propagation delay) as mentioned in [Alizadeh et al. (2010b)], we observe the results obtained in Figure 4.10. We see that both the NeSt 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 our analytical modeling and hence corroborates the accuracy of NeSt's implementation of our proposed mechanism.

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 4.11 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 4.5.3's simulation setup. In Figure 4.11a we observed that for a mean RTT of 25, it took close to 50 RTT intervals (or 1.2ms) for Q_i

to converge. Similarly for RTT values of 62 and 98 s in Figure 4.11b and 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 1-3ms. It is to be noted that not all applications take this long to converge - for example, in Figure 4.11a, 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 15ms 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.

(a) (b)

Figure 4.12: Convergence comparison between (a) DCTCP and (b) QTCP

In a datacenter environment it is not realistic to assume that all ows start at the same time. Hence in Figure 4.12 we also considered a scenario where multiple ows 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 98s. We noticed that QTCP converged faster than DCTCP in Figure 4.12a by sim 5ms i.e., by more than 50RTT slots. This shows that QTCP provides QoS differentiation faster than DCTCP provides fair treatment. In our previous work [Biswas

(b)

(a)

Figure 4.13: (a) Workload distribution along with (b) evaluation of a scenario with storage bottleneck

et al. (2021)], we also looked into the behavior of applications with differing RTT values and showed that DCTCP is biased towards ows 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 [Alizadeh et al. (2011)]; we have also observed this in our simulation.

Evaluation of Throughput Sensitive Applications

We now look into the behavior and effect of QTCP in an E2E context. We utilize NeSt to create the simulation setup shown in Figure 4.8 with the network links capacity being 100Gbps. In this testbed the storage end consists of 10 storage servers. Each storage server contains a simulated 1TB NVMe SSD which simulates the Scenario 3 QoS differentiation mentioned in 4.4.2. The applications generate storage traf c following an exponential distribution, targeted 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 signi cance - 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 90Gbps, 60Gbps and 30Gbps of write traf c over the 100Gbps 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 4.13a 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 4.14 we compare the throughput differentiation provided

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

Figure 4.14: E2E Behavior of a Write Only Workload

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 traf c in an E2E context. For write-only traf c, the bottleneck arises on the sender side. We notice in Figure 4.14a

that for DCTCP, all three receive close to identical treatment, i.e., the observed throughput for the three applications is close to 33Gbps. This is because DCTCP treats all ows the same during a congestion episode. It is to be noted that the Low Class application pushes only 30Gbps of traf c and hence receives the entirety of its required bandwidth (due to it being less than the DCTCP divided throughput of 33Gbps). 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 48Gbps, with Medium and Low receiving 31Gbps and 17Gbps respectively. In Figure 4.14b, 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 squeezing the Low class traf c 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 traf c. As explained before, the response side in Figure 4.14c 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 3. We noticed in Figure 4.13b 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 4.15. 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 30Gbps 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

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

Figure 4.15: E2E Behavior of a Read and Write Intensive Workload

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 4.15c) and writes (in Figure 4.15a). However, in the case of QTCP as the transport protocol, we observe for the sender side in Figure 4.15a, 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 4.15b again respects the QoS differentiated offered load due to no bottleneck. However, unlike Figure 4.14, in this case, the response side is a bottleneck too due to the existence of reads. Hence in Figure 4.15c QTCP provides the necessary QoS differentiation to the experienced bandwidth for each application.

For a read-only workload in Figure 4.16, the bottleneck is on the response side and we observe in Figure 4.16 that as expected - QTCP once again provides the required throughput differentiation, unlike DCTCP which doles out fair treatment to all QoS classes.

Figure 4.16: Comparison between DCTCP and QTCP for a read only workload

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 4.17a depicts this scenario where we have one ow per class present in the datacenter before other ows enter the overall traf c. We introduced another Low class application around the 50ms 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 4.4.1, the target throughput is assigned on a per class basis and not a per ow basis. Hence the introduction of a new ow pertaining to a QoS class only results in the target throughput of that class being shared between all ows of that class. We also see in Figure 4.17a that as soon as an application completes its requests, the other ows pertaining to that QoS class grab on to its assigned target throughput.

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

(a) (b)

Figure 4.17: Evaluating QTCP with (a) multiple intermittent ows and (b) interfering DCTCP traf c

nario. QTCP continuously measures the interference of the other ows and adjusts the bottleneck bandwidth accordingly. We assume that is known initially (can be given or even estimated using methods detailed in [Cardwell et al. (2016)]). If the aforementioned non-QoS sensitive ow alters the value of then each of the QoS sensitive ows recognizes this perturbation as detailed in Figure 4.18. The target throughput requirement of a ow i is de ned as $target_i$ and the measured throughput as $actual_i$. Quality Factor converges to its steady state value of 1 pretty fast (as explained in 4.5.3) and hence any variation in that value by a value greater than (for our experiments the value for is taken as 5% or 0.05) is understood to be caused by an interfering ow - either due to its interference with these ows 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 modi ed accordingly. In the given algorithm, the fraction of the bottleneck bandwidth assigned is de ned as $ratio_i$ and the current measured Quality Factor is depicted as Q_i. The current congestion window for ow i is W_i.

We evaluated the proposed modi cation in Figure 4.18, by simulating a scenario in Figure 4.17b where an interfering DCTCP ow enters and leaves in the middle of a simulation

```
W_i     W_i p1      {2q
if (Q_i     1:0 ¡     q)   // means there is interference  ow
          factor =      ;
else if(Q_i     1:0       )   //interference  ow left
          factor =   ;
elsefactor = 0.0
     p       factorq    target_i
target_i     ratio_i       ; Q_i     target_i {actual_i
if (Q_i     1) W_i     W_i Q_i
```

Figure 4.18: Window modulation in presence of interfering ows

involving three different applications pertaining to the 3 QoS classes. We can see in Figure 4.17b that as the DCTCP ow enters the traf c around the 75ms mark, the QoS ows back off and then again increase their measure ow rates when the interfering ow leaves around the 250ms mark. One could argue that reserving resources for the DCTCP ow can also be a solution, however, this would result in under-utilization of the network bandwidth when the DCTCP ow is not present.

Evaluation of Latency Sensitive Applications

Figure 4.19: Deadline Miss Comparison for DCTCP vs TCP vs QTCP

We now move on from throughput sensitive applications to look at latency sensitive applications. We rst 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 de ned as 5,6,8ms respectively. The mean transfer size of the traf c is 2MB while the ows follow a Poisson distribution with an average utilization of 80%. We compare the performance of QTCP with DCTCP and D^{2}TCP [Vamanan et al. (2012)] in Figure 4.19 - the latter is another QoS based window modulation mechanism which takes into consideration the “deadline” i.e., the target latency of each class/ ow. D^{2}TCP calculates a ratio d_i between the measured delay and the target delay for a i ow and modi es the window size using $_i^{d_i}$ instead of $_i$. 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 4.19 we compare two different situations - normal and stressed situations. The former situation deals with High and Medium applications generating traf c at a frequency lower than the others - 10% and 20% respectively while in the stressed scenario every class contributes equal load. We can see for the normal situation in Figure 4.19 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 traf c 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 traf c 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 episodes in the case of latency sensitive applications is assumed to be transient and eeting in nature. This is due to the fact that in the case of longstanding congestion episodes, 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 traf c, we consider simulations lasting 1-5s, which depicts the time

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

Figure 4.20: Tail Latency comparison when QoS treatment is not coordinated

period of a single congestion episode.

Keeping our NeSt setup the same as in 4.5.3, 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 40Gbps. This creates a stressed scenario with bottleneck both in the network (network link is 100Gbps) and in the storage end. From our discussion in 4.3.3 we know that writes are expensive operations in SSD which in turn can lead to high tail latency values. We rst look into the need for consistent coordinated E2E QoS treatment. If the storage end is oblivious of the E2E QoS requirement, it may provide its own notion of QoS i.e. QoS at a local level, which is inconsistent with the required QoS con guration. For example, the FTL in the device may use request size as a metric to determine QoS class of incoming traf c. In Figure 4.20 we assume a scenario where the storage end is unaware of the QoS differentiation provided by the network. This helps us in determining both the effect of storage QoS on E2E QoS and also the need for coordinated QoS treatment. In spite of storage latency outperforming the network latency, variability in storage latency (caused due to write induced high tail latencies) can affect E2E latency differentiation. We hence consider 3 applications App1, App2 and App3 which QTCP in the network treats 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 both in Figure 4.20a and Figure 4.20b. App 1 is treated the worst in spite of it requiring the best treatment. We remedy this by ensuring both network and storage provide consistent coordinated QoS treatment in our subsequent experiments.

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

Figure 4.21: Tail Latency Comparison

In Figure 4.21 we observe the aforementioned write triggered large tail latencies for the Low class application, where the 50th to 99th percentile latency value ranges from 0.7ms to 1.5ms. This is extremely poor performance for a device whose median latency values are supposed to be less than 100s. However, it is to be noted that the storage level QoS differentiation mentioned in 4.4.2 ensures that while the Low QoS class suffers, High and Medium report values close to 60s and 100s 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 4.21b, fails to provide adequate differentiation while QTCP helps in providing adequate differentiation between the three classes in Figure 4.21c. 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 signi cantly poorly due to the stressed scenario and hence it performs the worst even in an E2E context with DCTCP.

Evaluation of RDMA Traf c

As discussed in 4.3.4 and 4.4.1, the feedback-based ow control in DCQCN works nearly identically to DCTCP. This means that the fundamental workings of QRDMA (i.e., QoS-aware DCQCN) and QTCP remain the same. This behavior was corroborated in our previous work [Biswas et al. (2021)], where we observed that for throughput sensitive applications, DCQCN treats all the ows equally during the congestion episode while QRDMA differentiates based on their offered load ratios. Hence we focus on evaluating latency sensitive storage ows utilizing DCQCN or QRDMA as its transport.

In Figure 4.22a we evaluated latency sensitive applications for QRDMA in comparison to DCQCN. The evaluation considers a stressed scenario similar to Figure 4.19. It is observed in Figure 4.22a for this case that QRDMA outperforms DCQCN with QRDMA having fewer deadline misses in comparison to DCQCN by 43-80%. However, QRDMA's latency deadline misses are more than QTCP's in Figure 4.19. This is because QRDMA is based on DCQCN and DCQCN requires parameter tuning. We have used the default parameter settings as noted in [Zhu et al. (2015)] because parameter tuning is out of the scope of this chapter. Further, from our discussion in 4.4.1, we know that remote PM traf- c utilizes QRDMA as a transport. It is unrealistic to assume that said PM traf c exists in isolation in a datacenter environment. Due to this need for coexisting DCTCP and DCQCN traf c (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 rst discuss the need for our Quality Factor metric in such a mixed traf c scenario.

We rst look into the bandwidth sharing between DCQCN and DCTCP followed by QTCP and DCQCN in Figure 4.22b and c. We utilize the topology shown in Figure 4.23a where a bottlenecked 100Gb/s link is shared by DCTCP and DCQCN traf c. Fig 4.22b

(a) (b) (c)

Figure 4.22: Bandwidth sharing comparison between (a) DCQCN vs DCTCP and (b) QTCP vs DCQCN

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 de ne it as R_{AI}, during the congestion recovery phase, current rate RC_i for DCTCP would be $RC_i \quad RC_i \quad R_{AI}$. On the other hand, DCQCN follows a fast recovery phase which essentially increases its current rate RC_i by approximately $RC_i \quad \frac{R_{AI}}{2}$, which is less than the DCTCP case. Hence, initially DCQCN ows grab much of the bandwidth but during fast recovery, DCTCP rate increase is higher than the DCQCN. Thus DCTCP ows gradually end up grabbing most of the bandwidth and stabilizes. Our experiment in Figure 4.22b corroborates this analysis. This is a result of DCQCN's optimal parameter settings which keeps queue depth below the threshold.

The QoS based adjustment of transmission rates using the Quality Factor metric xes this problem as shown in Figure 4.22c and both get equal bandwidth under congestion. Note that original DCQCN suggests bandwidth reservation for the TCP traf c, in order to provide fairness for DCQCN ows [Zhu et al. (2015)]. But that may lead to under-

(a) (b) (c)

Figure 4.23: (a)Evaluation setup for coexisting ows and bandwidth sharing comparison between (b) QTCP w DCQCN ows and (c) QRDMA w DCTCP ows

utilization during non-congestion period.

Notice that the notion of quality factor can also be used to differentiate QoS for individual ows. Using the same topology as Figure 4.23a for our evaluation in Figure 4.23a, we set the relative ratio of ows as 1: 1.5: 2: 3. We observe the QoS differentiation within the QTCP ows while also making way for the remaining DCQCN ows. We can see similar behavior for DCTCP with QRDMA ows offering the same relative ratio of offered load in Figure 4.23b. The ECN parameter setup is in favor of the optimal performance of DCQCN, not QTCP, thus this makes respecting the target throughput dif cult in Figure 4.23a. However, in the case of standalone DCTCP ows with QRDMA ows in Figure 4.23b, the ratio is approximately equal to the target throughput ratio.

4.5.4 Evaluation of Mixed PM and Storage Traf c

It is to be noted that for practical reasons, only one of QTCP or QRDMA is likely to be con gured or enabled in the datacenter unless there is a real need to have both. Mixed PM (which is best served by QRDMA), and storage traf c (which is then better carried by QTCP) is one such scenario. A mixed QTCP and QRDMA storage traf c scenario may also arise but that is beyond the scope of this work. In this section, we evaluate the behavior of small-sized PM traf c (using QRDMA) sharing the same bottleneck link as the larger-sized storage traf c (using QTCP). For this experiment, the PM traf c is Poisson with a

size of 128B or 256B with equal probability while the storage access size follows the same distribution as in Figure 4.13a. Also, Persistent Memory over Fabrics (PMoF) [Grun et al. (2018)] using RDMA is another solution to ef ciently 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 traf c scenarios - one with read-only traf c and one with write-only traf c in Figure 4.24a and Figure 4.24b respectively. The accompanying workloads follow the same distribution as in 4.5.3. We observe that for both types of workloads, having no in-network strict priority for the PM traf c results in it experiencing E2E latency values greater than 20s. This is undesirable due to the low latency requirement of the PM traf c. We can see that the PM access latency itself is in the range of 300-800ns. Grun et al [Grun et al. (2018)] talk about how we can attain close to 3ms in E2E latency for PM traf c by using RDMA. Figure 4.24 shows similar performance with QRDMA as the transport. We see that our in-network prioritization (proposed in 4.4.1) results in 82% improvement, with the E2E latency ranging from 3-4s consistently. This is despite the presence of other QTCP traf c i.e., High, Med, Low class QoS sensitive traf c. As mentioned in 4.4.2, 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.

Effect on Throughput Sensitive Flows

In 4.4.1 we talk about buffer reservation for PM traf c packets so as to make way for strict priority small transfer memory traf c. This in turn brings forth the question as to what effect this reservation has on throughput sensitive large transfer traf c. We observe in Figure 4.25 the treatment received by the throughput sensitive QTCP storage ows (High, Medium and Low classes) in the background for the evaluation shown in Figure 4.24. The bottleneck link is 100Gbps while the offered load for the three large transfer applications

(a) (b)

Figure 4.24: Comparison of E2E Latency for PM traf c for (a) Read Only and (b) Write Only Workload

are 90Gbps, 60Gbps, 30Gbps i.e., a ratio of 3:2:1. We show only the bottlenecked response path for the read only workload in Figure 4.25a and the bottlenecked request path for the write only workload in Figure 4.25b.

(a) (b)

Figure 4.25: PM Traf c Effect on Throughput Sensitive Applications for (a) Read Only and (b) Write Only Workload

In Figure 4.25 we see that the effect of the PM traf c is negligible due to its small request size. Not having to make way for the small transfer traf c (when QTCP is in isolation) in Figure 4.25a and Figure 4.25b only increases the throughput of each ow by

(a) (b)

Figure 4.26: Effect of PM traf c with (a) different intensities and (b) incast degree 1:12

close to 1Gbps. It is also to be noted that in spite of the presence of the strict priority memory traf c, the throughput sensitive traf c still receives differentiation in the offered ratio due to the use of QTCP.

Similarly, we observe the behavior of all three types of applications in Figure 4.26a with varying intensities (10%, 12%, 15% and 20%) of PM traf c. Even in this case the carried throughput reduces slightly but QTCP ensures the QoS differentiation for the rest of the ows according to their target ratio. We nally test the bandwidth distribution of throughput sensitive ows with a different incast degree (1:12) in Figure 4.26b, for a PM traf c 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.

Effect on Latency Sensitive Storage Flows

We now look at the E2E behavior of a mixed traf c scenario comprising of PM traf c and latency sensitive large transfer storage traf c in Figure 4.27. We keep the same simulation setup along with the same workloads as discussed in our E2E evaluation in 4.5.3. We evaluated the tail latency for all 4 of the applications in two separate scenarios to note the gains we make with our proposed changes. Similar to our evaluation in Figure 4.24, the

rst scenario considers the four ows with the PM traf c being offered no strict priority while the second scenario is the same with strict priority. In Figure 4.27a, we observe similar behavior to what we observed in Figure 4.24. However, it is interesting to note that with in-network priority, the latency observed is fairly consistent within the range of 3-4 s across the 50th, 90th and 99th percentile. However, when there is no in-network priority, the latency increases from 15 s to 23 s. Thus our proposed change improves PM tail latency by 83%. We also observed that having strict priority for PM traf c slightly increases

(a) PM Traf c

(b) Storage Flows w/o PM priority

(c) Storage Flows w PM priority

Figure 4.27: Tail Latency comparison of mixed traf c

the observed latency for other latency sensitive storage ows in Figure 4.27c as compared to Figure 4.27b. For larger percentile values, the increase in latency was minimal. For example, the High QoS class latency value changed from 10ms to close to 11ms for the 90th percentile. Similarly, for the Medium class it increased from 13ms to 15.3ms. The increase in latency is expected as there is no free lunch i.e., strict buffer reservation for one ow (or even the existence of another ow) in a congested scenario would always hamper the performance of other ows. But the QoS differentiation is still maintained in spite of the slight increase in latency due to QTCP's stable reaction to background traf c.

4.6 Related Work

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

4.6.1 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 DRAM caching at the level of device (e.g., a disk controller cache), storage server, host (discussed in 4.3.1), etc. All of these modules provide an opportunity of QoS differentiation in order to intelligently manage the access latency; however, QoS differentiation in these contexts is not widely researched or practiced. He et. al talk [He et al. (2014)] about a caching methodology that utilizes applications' SLA metrics (ex., 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 [Gupta and Kant (2020)] 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. Prabhakar et al [Prabhakar et al. (2011)] proposes a dual step QoS aware cache management mechanism for multi server environments. The rst step breaks down the application level QoS requirements to sub-QoS components expected by the storage server, followed by a feedback control loop-based storage allocation step to determine cache space allocation depending on QoS metrics.

ElNably et al. [Elnably et al. (2012)] 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 [Elnably et al. (2012)] and [Elnably and Varman (2012)]. 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 [Peng and Varman (2020)] 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 ILP (Integer Linear Programming). On the other hand, Kim et al. [Kim et al. (2020)] 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 [Gugnani et al. (2018)] design a QoS-aware NVMe emulator which provides support for weighted round robin and de cit round robin arbitration mechanisms. However, none of these works delve into the QoS of the requests once they enter the device. Also, these works do not discuss network QoS that is relevant for differentiated, end-to-end ow control.

4.6.2 Network QoS

Network QoS is a deeply examined topic, especially in the context of Ethernet-based networks, however, much of it concerns the WAN, where packet drops are routine and the packet drop rate forms an important QoS parameter. Since our focus is end-to-end 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 ows, some variants address the differentiated treatment. Differentiated treatment in the Transport layer can be classi ed into two categories:

Credit Based Control: Expresspass [Cho et al. (2017)], Re ex [Klimovic et al. (2017)] 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- ow 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 [Zhu et al. (2016, 2014)] the authors have focused on QoS aware ow 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 ow control in the TCP context. Homa [Montazeri et al. (2018)], L^2DCT [Munir et al. (2013)], D^2TCP [Vamanan et al. (2012)], PDQ [Hong et al. (2012)], D^3 [Wilson et al. (2011a)] consider QoS in terms of individual ow completion time (i.e., deadline). Homa [Montazeri et al. (2018)] tackles the head-of-the-line (HoL) blocking problem that TCP streams present. It uses in-network queue priority to give small messages (¡1000 bytes) low latency. However, it does not consider the effect of such in-network queuing on other QoS sensitive traf c. In a real-world scenario, latency sensitive small transfer traf c does not exist in isolation. L^2DCT [Munir et al. (2013)] and D^2TCP [Vamanan et al. (2012)] on the other hand adjust the TCP congestion window for various ows based on the stated QoS parameter. One of the main problems with these methods is that to de ne 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 ows. PDQ [Hong et al. (2012)] proposes distributed scheduling algorithm, where the switches coordinate among themselves to schedule the high priority ow earlier (i.e., the ow 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 [Wilson et al. (2011b)] needs speci c 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.

4.7 Conclusions

In this chapter, we discuss the end-to-end QoS 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 traf c in the network and

show how the PM traf c can achieve the very low latency that it required without the need for any specialized hardware. We demonstrate that a consistent con guration 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 traf c that uses a low latency RDMA based transport, where multiple QoS classes are de ned for the storage traf c that uses TCP as the transport. In the future, we will consider multiple QoS classes of RDMA based traf c as well.

CHAPTER 5

NEST : A QOS DIFFERENTIATING END-TO-END NETWORKED STORAGE SIMULATOR

5.1 Overview

Emerging Non Volatile Storage devices today utilize the Nonvolatile Memory Express (NVMe) protocol to meet their high throughput and low latency needs. In a datacenter environment, applications accessing multiple such devices over the fabric (i.e. the network) tend to have Quality of Service (QoS) requirements pertaining to offered throughput and experienced latency. Satisfying these requirements in an end-to-end fashion (starting from the host application to the device endpoint via the network and the storage access protocol) becomes increasingly dif cult due to bottlenecks that may arise at various points. Additionally, the expense associated with recreating such an environment or even for accurately testing end-to-end proof-of-concept hypotheses is dif cult to bear in an academic context. In this chapter we propose NeSt - a QoS Differentiating End-to-end Remote Storage simulator. NeSt simulates a datacenter environment with multiple high throughput and low latency nonvolatile devices accessed via the NVMe-over fabric protocol. It does so by combining the highly detailed network simulator NS3 with enhanced modules of the comprehensive Solid State Drive simulator - MQSim. Additionally, it simulates a realistic scenario with multiple storage applications accessing said devices via the network by utilizing real published storage traces. We evaluated NeSt in comparison to both real world scenarios and against existing simulators along with testing its individual features in isolation. We observed that NeSt's accuracy in simulating a real scenario is approximately 92.5%. To the best of our knowledge, NeSt is the rst such simulator to accurately present such a detailed view of an end-to-end networked storage environment.

5.2 Introduction

Storage systems in present-day data centers are organized as storage servers, with each of these servers being made up of both storage and memory devices, ex. Solid Sate Drives (SSDs) and Storage Class Memory (SCM) devices. Remote applications accessing these devices (via storage access protocols such as Nonvolatile Memory Express over Fabric, i.e. NVME-oF) have been requiring increasingly high throughput and low latencies for its requests. In order to respect these requirements, it becomes signi cantly important to provide speci c Quality of Service (QoS) treatment across the request's end-to-end (E2E) path, where the end-to-end path broadly comprises of four components - the host system, the request and response path to carry the requests, the storage access protocol (that allows interfacing with the storage device, ex. NVMe-oF) and the storage device itself. Capturing a request's E2E journey (i.e. starting from the host system and back to the host system) in a datacenter environment comprising of multiple hosts and multiple storage servers is a complex task. This work presents a tool (i.e. a simulator) that accurately captures the E2E network path along with a comprehensive target storage endpoint, while also modeling the aforementioned datacenter environment.

The growth in network performance has not mirrored the exponential improvement in storage device performance. For example, datacenter SSDs today can drive traf c up to 40-45Gbps. This means that a few of such devices can be more than enough to cause congestion in a traditional 100Gbps network link. Hence attaining differentiation in the network for applications with QoS requirements is of paramount importance. Similarly, latencies in the storage device can dominate E2E latencies in certain cases due to the internal architecture of these devices. This can lead to long tail latencies (i.e. 90th percentile latency values or above) which in turn could complicate QoS differentiation. Recreating these scenarios or testing proof-of-concept hypotheses is a dif cult problem to solve due to the lack of proper E2E simulation tools in the open source community. Current solutions

aim to model only a speci c component in the E2E path, for ex. the network (via network simulators such as NS3) or the storage device (ex. MQSim and SimpleSSD). Network simulators aim to model the performance of individual packets in the datacenter but fail to simulate the target endpoint, i.e. the device or the storage access protocol. Similarly, storage simulators model the performance of single storage devices, ex. SSDs while failing to consider that applications accessing this device may be remote or that application requests may span multiple devices. This is a common issue in the datacenters where the storage volume is typically presented to applications as a single virtual entity, when in fact it may be mapped multiple storage devices underneath it. For ex., as mentioned in Chapter1, host requests to a storage volume may be mapped to a different storage server. Similarly, responses from the storage end can be generated from multiple storage servers. For example, Hadoop applications have the map reduce phase that can generate vast amounts of data from multiple storage servers. Scenarios such as these are dif cult to simulate due to the lack of a comprehensive endpoint architecture in network simulators and the lack of multi device simulations (along with no network component) in storage simulators. We utilize the robust architecture of existing network and storage simulators and extend them to address these issues, while also adding relevant features wherever required in our proposed tool.

The rest of this chapter is organized as follows - Section5.3 talks about the background and motivation of this work along with the limitations of existing works. Section 5.4 discusses our proposed simulator - NeSt along with its features and the implementation efforts we undertook. Section 5.6 evaluates NeSt by comparing to existing open sourced simulation tools and by testing its various features. Finally the chapter is concluded in 5.7.

5.3 Background and Related Work

5.3.1 Storage End QoS

The current SSD technology has made a quantum leap over the traditional HDDs both in terms of access latency and throughput. For example, even the rather inexpensive SSDs available today are capable of driving up to 25-35 Gb/sec IO throughput Davis (2016) with nominal latencies of under 100 s Tallis (2019). The newer storage technologies can provide even lower latencies, e.g., 10-20 s Zheng et al. (2019) latency for Intel's Optane technology[1] and Kioxia's XL- ash technology. Such performance levels have profound implications at many levels, the most basic of which is the interfacing of storage devices. In particular, the storage access protocols have to not only support increasing levels of throughput but also provide low latency, so that the protocol latency does not add significantly to the storage access latency. The NVMe protocol NVMe (2019) has recently become the dominant protocol due to its ability to support high throughput and low latency. It achieves the latter via an improved queuing interface on top of PCIe interface, and enables the device to interact with the CPU directly instead of the indirect DMA interface used by older protocols like SATA and SAS. NVMe interface supports a large number of “queue pairs” (request submission and completion queues). These queue pairs can be assigned one of the following priorities (urgent, high, medium, low), with weighted round-robin (WRR) scheduling within a priority class. This provides a rich mechanism to support QoS at the level of storage access handling.

Extending the storage access over the network requires the local access protocol such as the NVMe to be transported from host to the target without changing the basic access semantics. NVMe over Fabric (NVMe-oF) NVMeOF (2019) provides this capability by encapsulating the submission and completion queue entries into transport-independent “capsules” for transfer between host and the target. The capsule transport also puts the

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

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 a reliable transport; two suitable choices for the transport protocol in data centers are DCTCP (based on TCP) and DCQCN (based on RDMA). The key requirement of data center transport protocols is the avoidance of any packet losses and low latency. Both DCTCP and DCQCN provide lossless transport, with the latter potentially providing lower latency, but neither provides any QoS differentiation.

5.3.2 Storage vs. Persistent Memory

The emergence of Storage-Class Memory (SCM) technologies has ushered in a new era where the very attributes that make them attractive for storage can also be harnessed to replace or augment traditional volatile memory. The exceptional characteristics of SCM, including very low-latency and high-throughput capabilities, make it a compelling option for memory solutions. However, the introduction of persistence into the memory hierarchy raises several additional issues related to data consistency.

While SCM technologies, such as Intel Optane[2], are currently somewhat slower than the venerable Dynamic Random-Access Memory (DRAM), they become even more appealing when their cost-effectiveness is factored in. Leading tech giants like Amazon and Oracle have recognized the potential of SCM and incorporated it into their offerings. For instance, Amazon ElastiCache[3] and Oracle's Persistent Memory Database leverage Persistent Memory (pmem) technology for data storage and database solutions. SCM devices can serve dual purposes, operating with either storage or memory semantics. Under storage semantics, access sizes are typically larger (e.g., 4KB), and threads requesting device access may be temporarily switched out until the desired data is received. Conversely, in

[2] https://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory/optane-dc-persistent-memory.html

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

the memory model, access sizes typically involve fetching a few cache lines, causing the CPU to stall until the data arrives. In this work we consider the storage model pertaining to SCM devices. Chapter4 explores the use of the memory model.

While locally installed SCM often bene ts from a DRAM-like interface (e.g., DDR) to provide low latency, the scenario changes when accessing it over a network. In this case, additional latency can be introduced, particularly when this traf c competes with other classes of traf c in the network. Given these complexities, it becomes essential to provide urgent Quality of Service (QoS) to it. Crucially, during congestion episodes, throttling this traf c is not desirable. This requirement extends beyond the endpoints of the network. The network switches themselves must be equipped to provide preference to this traf c, ensuring that it remains a high-priority entity within the network architecture. Through network-level intervention, it is possible to reduce its latency substantially, optimizing the utilization of this promising technology in the context of error diagnosis and microservices architecture.

5.3.3 QoS Limitations in Current Data Center Networks

Modern data center networks face a multitude of challenges in achieving ef cient Quality of Service (QoS), particularly when catering to a diverse array of traf c classes. One of the prevailing hurdles is the dominance of kernel-centric implementations, often resulting in undesirable latency costs for certain types of traf c with stringent low-latency requirements, such as Persistent Memory (PM) traf c, demanding response times of less than 10 microseconds. To address these latency-sensitive needs, novel solutions are essential. The NVMe speci cation, for instance, introduces the concept of a distinct region known as the Persistent Memory Region (PMR), offering memory-level data read and write speeds. In conjunction with the use of Remote Direct Memory Access (RDMA)-based solutions as the transport layer mechanism, known for its hardware support and reduced reliance on kernel operations, the handling of small transfer memory traf c is signi cantly improved. How-

ever, it's important to note that conventional Data Center TCP (DCTCP) remains a viable choice for handling large-scale storage traf c. Ef ciently providing preferential treatment to different classes of traf c within the network hinges on two core aspects:

(a) Differentiated Scheduling: This aspect involves reducing the latency for traf c belonging to a high-priority QoS class compared to other classes during standard traf c conditions. Implementing appropriate scheduling can be readily accomplished at the end-points, and emerging Smart Network Interface Cards (NICs) often possess the capability to perform this in hardware. Nevertheless, switches in the network infrastructure must also accommodate such scheduling, necessitating the inclusion of class designations in the packets, typically indicated using the Ethernet “class of service” (CoS) at Layer 2.

(b) Differentiated Congestion Control: To achieve the desired relative behavior, such as a speci ed proportion of available bandwidth, across various traf c classes, differentiated congestion control mechanisms are essential. The transmitter can implement these mechanisms, but they require feedback regarding network congestion.

In the context of Ethernet and IP-based networks, Quality of Service (QoS) has been a subject of extensive study. Standardized mechanisms like Differentiated Services Code Point (DSCP) have been widely adopted, particularly at the IP layer. However, DSCP was primarily designed for wide area networks, where transport protocols, such as conventional TCP, are sensitive to packet losses, making a packet loss-centric QoS treatment reasonable. In contrast, high-speed data center networks prioritize avoiding packet losses, rendering DSCP less suitable. Moreover, DSCP operates on a “hop-by-hop” basis within each router, rather than providing uniform end-to-end treatment. To achieve consistent end-to-end QoS treatment, data center networks are turning to loss-free transport layers, notably Data Center TCP (DCTCP) on the TCP side and Data Center Quality of Service Congestion Noti- cation (DCQCN) on the RDMA side. These technologies embrace proactive congestion control mechanisms, where senders monitor congestion through feedback mechanisms like Explicit Congestion Noti cation (ECN). ECN employs two bits in packet headers, one sig-

naling congestion in the forward path, and the other set by the receiver in packets traveling in the opposite direction.

A key distinction emerges in the realm of proactive congestion control: DCTCP uses ECN to detect queue buildup in switch buffers (as opposed to actual packet drops) and modulates ow rates accordingly. Similarly, Remote Direct Memory Access over Converged Ethernet Version 2 (ROCEv2) implements the RDMA protocol atop the IP layer and employs a related mechanism known as DCQCN for proactive congestion control. However, it's worth noting that neither DCTCP nor DCQCN inherently possess QoS awareness, often providing non-differentiated treatment to all ows when congestion is detected. Hence addressing these challenges to achieve re ned QoS and low-latency support for speci c traf c classes remains a crucial focus for optimizing performance and reliability.

5.3.4 Overview of DCTCP and DCQCN

The Explicit Congestion Noti cation (ECN) is an optional feature in TCP/IP which is used to provide end-to-end noti cation for network congestion without dropping packets. It utilizes the two right most bits in the Differentiated Services eld of the IP header to notify congestion to the receiver endpoint while the ECN Echo or “ECE” bit (a reserved bit in the TCP header) is used to provide congestion feedback to the sender endpoint.

We now discuss the working principles of DCTCP and DCQCN as these are the two suitable transport layer protocols for data centers, based on TCP and RDMA respectively. Both DCTCP and DCQCN use exponential smoothing to vary the congestion window size (CWND) with DCTCP using the ECE bit to mark acknowledgments so as to denote that congestion has been detected. A fraction of arriving acknowledgments (with ECE bits set) in a window is used as the control parameter in the smoothing process. That is, for competing TCP connections (or ows), let us consider q_i as the aforementioned fraction of acknowledgments for the ith ow, having set ECE bits during the nth window. We can

hence exponentially smooth this value $f_i(n)$ over succeeding windows as follows:

$$\alpha_i(n+1) = (1-g)\alpha_i(n-1) + g f_i(n-1) \tag{5.1}$$

where α_i is the exponentially smoothed value and $0 < g < 1$ is the smoothing constant.

DCTCP algorithm: DCTCP reduces the window per round-trip time (RTT) in proportion to the latest estimate of f_i such that in the limiting case of $f_i = 1$, the window is halved. Hence the window controlling mechanism is formulated as follows:

$$W_i(n+1) = W_i(n-1)\left(1 - \frac{\alpha_i}{2}\right) \tag{5.2}$$

DCQCN algorithm: DCTCP, however, is a solution for TCP and hence would not work for RDMA transport mechanism. This brought forth the need for an RDMA speci c solution - Data Center Quantized Congestion Noti cation (DCQCN). It utilizes the PFC feature in data center Ethernet. PFC (which is an L2 QoS capability) however, cannot distinguish between ows and paths and stops all incoming ows whenever congestion is detected in a path even if no congestion is present in other subsequent destination paths. A PAUSE frame is generated when congestion occurs, and this stops all ows. DCQCN uses a combination of both PFC and ECN to circumvent this issue. However, since ROCEv2 utilizes a connectionless protocol i.e. UDP, it cannot leverage the ACK packets like in DCTCP. Hence the congestion feedback mechanism needs to be request-agnostic. The switches mark the ECN bits once congestion is detected, which the receiver NIC detects. As there are no ACKs to send for every ECN marked packet received, the receiver NIC sends back a ROCEv2 standardized Congestion Noti cation Packet (CNP) to the sender NIC for every N microsecond interval . This goes on for every interval until it stops receiving ECN-marked packets. As the maximum number of CNP received during each interval is 1, Eq.5.1 is updated for DCQCN as follows,

$$\alpha_i(n+1) = (1-g)\alpha_i(n-1) + g \tag{5.3}$$

The sender modi es the rate reduction factor according to Eq.5.3 and also stores the current rate so as to utilize it later on for ow rate recovery. It is formulated as follows,

$$RT_i(n+1) = RC_i(n) \tag{5.4}$$

$$RT_i(n+1) = RC_i(n) \cdot (1 - \frac{\alpha_i}{2}) \tag{5.5}$$

The stored rate value of $RT_i(n+1)$ is used to increase the ow rate once congestion episode passes. There are two phases for increasing ow rate - fast recovery and additive increase. During fast recovery, the ow rate RC_i is rapidly increased so as to reach the target value of $RT_i(n+1)$ in successive iterations, i.e. :

$$RC_i(n+1) = \frac{RT_i(n) + RC_i(n)}{2} \tag{5.6}$$

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

$$RT_i(n+1) = RT_i(n) + R_{AI} \tag{5.7}$$

$$RC_i(n+1) = \frac{RT_i(n) + RC_i(n)}{2} \tag{5.8}$$

Despite their bene ts in proactive congestion control, neither DCTCP nor DCQCN possesses any capability for service differentiation. This results in all ows being treated equally during a congestion episode i.e. bandwidth or other QoS metrics are divided equally between all ows irrespective of their QoS requirements. For example, during a congestion episode, a high priority ow is treated the same way as a low priority ow, thus causing more harm to the former. Similarly, for emerging persistent memory ows, they should be given urgent priority as one of the fundamentals of memory traf c is it comprises low latency and delay sensitive ows. Treating these ows the same way as other remote storage traf c results is not desirable. It is also to be noted that this lacking in the transport level renders QoS differentiation solutions in NVMe storage protocol (queue arbitration, block layer ow control Guz et al. (2017b)) moot.

Figure 5.1: Architecture of MQSim

5.3.5 Network and Storage Simulators

In the realm of network and storage systems research, various simulators have gained prominence for their vital role in modeling and evaluating complex infrastructures. Among these, NS3 stands out as a versatile network simulator capable of faithfully replicating both wired and wireless networks, while OMNeT++ is a discrete event simulation framework known for its pro ciency in modeling communication networks. For network professionals, GNS3 and Cisco Packet Tracer offer powerful tools for network emulation and visualization, aiding in network design and education. In the domain of storage simulation, tools like FIO, DiskSim, and Iometer are widely used for benchmarking storage device performance. OpenIO and Ceph Simulator are instrumental for assessing the capabilities of distributed storage solutions. These simulators collectively contribute to comprehensive research, development in network and storage systems, offering insights into behavior and performance in controlled environments. Speci cally, MQSim, introduced at USENIX FAST 2018, serves as a specialized simulator designed to simulate a Solid-State Drive (SSD). It replicates the intricate internal structure of modern NVMe SSDs (shown in Fig-

ure 5.1) each dedicated to simulating different facets of the SSD. These key components encompass:

- The host module, responsible for facilitating communication between host applications and the SSD device, offering the capability to handle storage trace reading and generate synthetic traces.
- The simulator includes the PCIe link and switch, featuring submission and completion queue pairs, enabling a realistic interaction between the host system and the SSD.
- Within the SSD module, the host interface establishes a connection to the PCIe link, taking charge of requests from the submission queue and managing completed requests within the completion queue.
- The SSD's DRAM buffer functions as a data cache, with requests routed to the Flash Translation Layer (FTL) in cases where the required data is absent.
- The FTL module closely emulates the functionality of a real SSD's FTL, incorporating tasks such as address translation, ash block management, and the management of background processes like garbage collection.
- The nonvolatile memory module situated in the SSD's backend includes planes, chips, blocks, dies, and pages, with the FTL playing a pivotal role in ensuring precise data location within this backend.

Empirical testing has revealed that MQSim closely approximates the performance of actual SSDs, with response time errors averaging between 11% and 18%. However, it is crucial to acknowledge the limitations inherent to MQSim, including incomplete support for NVMe's Weighted Round Robin queue arbitration mechanism, its inability to simulate multiple devices, and the absence of support for Storage-Class Memory (SCM) technologies and remote device access. These limitations led to the development of NeSt, a framework speci cally engineered to address these issues.

Moving beyond the domain of SSD simulation, in the broader landscape of computer network research, network simulators play a pivotal role. NS3, a versatile network simulator developed using C++ and Python, has garnered substantial attention across both academic and industry domains over several decades. NS3 possesses the capability to simulate wireless and datacenter networks with a strong focus on accurately reproducing the entire network stack, inclusive of host applications. It has the feature to customize simulation code within the `main()` function of a user-implemented C++ file.

In the context of NS3, “nodes” represent endpoints, including senders and receivers, while “devices” signify physical components such as network interface cards and switches. The connections between these nodes and devices are established through “channels” that employ network protocols to manage communication. However, it's essential to note that, while NS3 excels in simulating datacenter networks, it does not extend its capabilities to simulate the storage component, as it primarily operates as a network simulator. Consequently, NS3 may not offer a complete representation of end-to-end scenarios within a datacenter environment, underscoring the importance of using specialized tools like MQSim and NeSt in tandem with general network simulators to comprehensively model complex systems.

Similarly, SimPy, implemented in the Python programming language, is a powerful and flexible discrete-event simulation framework that is widely used for modeling and simulating complex systems, including storage systems, network protocols, manufacturing processes, and more. Its architecture is designed to facilitate discrete-event simulation and provide a flexible framework for modeling complex systems. It comprises several key components:

- Environment: At the core of SimPy is the simulation environment, which manages the simulation timeline, schedules and executes events, and maintains a list of events for orderly processing.
- Processes: Entities and agents are represented as processes in SimPy. These pro-

cesses are implemented as Python generator functions and de ne the behavior of individual entities in the simulation. They create events, yield control back to the simulation environment for modeling waiting periods, and respond to events as they occur.

- Events: Events are integral to its architecture and represent occurrences or points in time where something happens within the simulation. Events trigger actions, update the system's state, and schedule future events. The environment maintains a queue of events to ensure chronological processing.

- Resources It allows for the modeling of resources that entities may compete for, such as shared network links or storage devices. Resources can be allocated and released by processes, enabling the simulation of resource contention scenarios.

However, there are some limitations. As SimPy is not intended to represent SCM systems. SimPy can be used to model storage-related operations like data access and transfer, but it does not have the tools like dynamic storage allocation and optimisation. Furthermore, Device-level Quality of Service may contain low-level details and interactions that are dif cult for SimPy to capture, particularly in hardware systems. Hence it is not ideal for simulating the complexities of hardware-level QoS since they are primarily intended for high-level event processing.

5.4 Proposed Simulator

In this section we discuss the implementation efforts we undertook and the features of NeSt. We do so by looking into individual features of note that NeSt supports.

5.4.1 End-to-End Path

An E2E storage access scenario (right from the host to the storage device and back to the host) follows the steps in given order:

1. Generate request from the host endpoint

Figure 5.2: High Level Architecture of NeSt

2. Break request into packets if it is a write request (i.e. write data) or into a single packet for a read request
3. Send request packets over the network
4. Receive request packets at the storage endpoint, assemble them (if write request) and push requests into device queues
5. Generate completed responses from the device
6. Break response into packets if it is a read request (i.e. requested read data) or into a single packet for the acknowledgement of a write completion
7. Send response packets over the network
8. Receive response packets, assemble them (if read request)

In order to accurately simulate this entire path, we need two major components:

- A: the path via which a read request (or write data) is sent towards the device endpoint
- B: the path via which the requested read data (or write completion acknowledgment) is sent back to the host endpoint

We do so by utilizing NS3's implementation of the network stack, including L2/L3 switches. NS3 utilizes the concept of nodes to simulate hosts with each host having the functionality to house different types of applications. The links connecting nodes are bidirectional and the transport used can be con gured. Supported transport options are datacenter versions of TCP, RDMA, UDP and QoS aware versions of TCP and RDMA (i.e., QTCP and QRDMA). The forward link is used for A to carry read requests or write data and the backward link is used for B to carry read data or write completion acknowledgements. The end point nodes mimic storage servers with multiple devices as explained in the subsequent subsection. We discuss more about QoS differentiation in the network in 5.4.5.

5.4.2 Multiple Devices with Multiple Con gurations

To simulate SSDs and devices using SCM technology, we utilized the widely cited SSD simulator - MQSim. As mentioned before, MQSim only simulates a single SSD. We made changes to the entire codebase of MQSim to ensure that it supports multiple devices. We did so by vectorizing all data structures, objects and function calls. Furthermore, we also enhanced MQSim to make sure that each device in the array of devices could support different con gurations. For example, we can simulate a SLC SSD, MLC SSD and even a TCL SSD at the same time. We carried this out by vectorizing the device con guration object present in MQSim. MQSim by default takes in a single con guration le as an execution parameter. After making the necessary changes, NeSt now provides the additional functionality of reading multiple con guration les during runtime (unlike MQSim which accepts the con guration le path as a execution argument). Further, the number of devices is a settable parameter. We have evaluated NeSt with a maximum of 30 devices (shown later in 5.6), however the number of devices can be increased depending on the speci cations of the system on which NeSt is run.

5.4.3 SCM Devices

As mentioned before, SCM devices are at least 3-5 times faster than traditional NVMe SSDs. They also do not possess the same internal architecture as NVMe SSDs. Hence, in our simulation of SCM devices, we switched off access to the SSD backend and extended the DRAM buffer module present inside MQSim (i.e. the buffer used inside an SSD to cache frequently accessed data) to resemble an SCM device. We changed the access latency parameters along with the capacity of the device to mimic the access latency behavior that is showcased by these devices in a realistic setting. A request that is tagged as Urgent is directed towards the SCM device. The request (whether it be a read request or write data) is resolved in our modi ed DRAM buffer cache and the response (i.e. read data or write acknowledgement) is sent out, thus resulting in an access' performance that is faster than SSDs and comparable to SCM devices. Our simulation of the SCM device allows it to be accessed both as a storage device (i.e. in request sizes ranging from 512 bytes to 4KB blocks) and as a byte addressable memory device (i.e. in request sizes reanging form 2-4 cacheline/64-128 bytes). Remote applications pertaining to the former mode of access are either throughput or latency sensitive due to their while the latter mode applies to ultra low latency sensitive applications speci cally. In our evaluation of the SCM device, we consider it as a storage device accessed by both throughput and latency sensitive applications. The overall end-to-end architecture of NeSt (with NVMe devices at the target end) is shown in Figure 5.2.

5.4.4 QoS Differentiation in the Storage End

Achieving QoS differentiation in the NVMe level is carried out by the widely used Weighted Round Robin (WRR) queue arbitration principle on the submission queues (i.e. the interfacing queues). Even though MQSim claims to support this feature, its implementation is a naive workaround of the actual mechanism. The original NVMe speci cation states that the queue arbitration occurs at the submission queues. However, MQSim

implements them in the chip level queues. Hence we made the necessary changes to accommodate this feature. We also preserve the existing WRR implementation in MQSim as it ensures that there is QoS differentiation inside the device as well, thus ensuring that the QoS differentiation is truly end-to-end. Hence the entire storage end architecture now includes WRR in the NVMe submission queues and even in the device level chip level queues. This helps in providing differentiation in the storage end between applications (as we show in our evaluations) but is not enough to achieve exact latency bounds due to the previously mentioned background activities in the SSD backend. Our previous works[Roy et al. (2021)][Roy et al. (2021)] have looked into tackling these background latencies and could be used in NeSt. However, that is currently beyond the scope of this tool.

5.4.5 QoS Differentiation in the Network

In this section we talk about how NeSt achieves QoS differentiation in the network. We look into our previous workJoyanta Biswas (2021) which tackled QoS differentiation in the network. It proposes a QoS aware transport solution built on the principles of DCTCP by introducing a ratio metric (called Quality Factor) to control the ow rate during congestion. The Quality Factor (QF) metric is calculated on a per class basis as a ratio between the target bandwidth and the measured bandwidth (or measured latency and target latency). A QF value greater than 1 indicates that slack is available and that the application belonging to this class needs to back off for other ows to meet their QoS targets. Consequently, a value less than 1 depicts that the QoS target has not been met and the ow waits for other applications to make way for it. All other window modulation equations are kept the same as DCTCP and in fact, in the absence of congestion, QTCP behaves exactly as DCTCP. The modi ed equations pertaining to QTCP are as follows:

$$W_i(n+1) = \begin{cases} W_i(n) + 1 & \text{No ECN} \\ W_i(n) + 1 - \frac{\alpha}{2} & \alpha_i \geq 0 \text{ and } Q_i \geq 1 \\ W_i(n) + 1 - \frac{\alpha}{2} Q_i & \alpha_i \geq 0 \text{ and } Q_i < 1 \end{cases} \tag{5.9}$$

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

We utilize this modified version of DCTCP (i.e. QTCP) in our E2E QoS differentiation as the transport protocol. However, DCTCP can also be used if the application does not require QoS differentiation. QRDMA is another transport protocol built on DCQCN (the RDMA version of DCTCP) to provide QoS differentiation to RDMA flows and can be used as a transport protocol too. However, its inclusion in NeSt is currently beyond its scope.

5.4.6 Integrating Simulators

In order to simulate the datacenter environment, we had to integrate the network simulator NS3 with our enhanced version of MQSim. We introduced a separate Interfacing Module (IM) which lies between NS3 and MQSim. The IM contains a pair of queues, i.e. the request and response queues. The former carries the write data and read requests towards the storage end while the latter carries the write acknowledgements and the read data back to the hosts.

The most significant issue involved with integrating two different discrete event simulators is their time synchronization. For example, let us consider 2 such simulators A and B with their own event tree and clock where the event tree stores every event that would occur at a certain time while the clock is used to trigger these events at the correct time. On completion of an event the event clock is progressed to the time of the next event in the event tree. Integrating the simulators A and B would imply that B has events dependent on A's events and vice versa. However, coordinating these events become difficult as both A and B have their own clock and tree and hence, in spite of the dependency, progress their clock to a future time once their own events are completed. Thus not waiting for the completion of events of the other simulator. We solved this issue by using only one of the simulator's clock and event tree. We leveraged NS3's event tree and clock by making modifications

in MQSim's simulation module module. These changes ensured that MQSim followed NS3's event clock and all its events were inserted in the NS3 event tree, thus preserving the relationship between dependant events.

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

5.5 Comparison with existing simulators

Figure 5.3: Comparison of Different Open Source Simulators

Open source simulators are widely utilized by the academic community to capture components crucial to a datacenter environment. For example, experiments involving datacenter networks can be accurately simulated using NS3 or Omnet++. Similarly, SimpleSSD or MQSim can be utilized for experiments involving simulation of the modern NVMe SSD. However, end-to-end evaluation of datacenter applications requires a simulator that combines all the necessary components together. CloudSim is an end-to-end cloud simulator that has been used by multiple research teams for datacenter simulations. However, it fails

to accurately simulate modern storage hierarchy in the datacenter, which includes SCM devices. Another limitation of these simulators is the lack of QoS support in an end-to-end context.

In Figure 5.3 we compare the features required for an end-to-end QoS differentiating datacenter storage simulator. For NS3, we can see that it does a good job of simulating the network, including QoS in the network (Layer 3 and Layer 4). However, it does not simulate the target device endpoint. Due to this component being absent in NS3, it also does not contain the functionality to replay storage block traces. Omnet++ also performs similar to NS3 in our feature comparison table. These limitations of both NS3 and Omnet++ are addressed in MQSim and SimpleSSD. However, both these SSD simulators do not consider accesses in a datacenter context and also do not support the simulation of multiple devices with varying con gurations. They also do not explore faster SCM devices.

CloudSim has been enhanced by research teams to support SSD accesses and thus in a way simulating end-to-end access of the target device. However, it does not simulate the storage access protocol (i.e., NVMe) and QoS differentiation is also absent. NeSt addresses these limitations of all the aforementioned simulators by supporting all the mentioned necessary features.

5.6 Evaluation

In this section we have carried out the E2E evaluation of different types of datacenter storage traf c while also comparing and testing the multiple features pertaining to NeSt.

5.6.1 Comparison with Real Systems

The rst step to evaluating a simulator is to compare its performance with a real environment scenario. This con rms its accuracy in capturing the said scenario. In this section we compare the latency observed in four scenarios with respect to SSD storage access -

- Local access to an NVMe SSD using a mixed read write workload generated using

Figure 5.4: Comparison between simulation and real environment

iperf

- Remote access to the same SSD (with NVMe-oF) using the same workload with TCP being used as the transport protocol protocol
- Simulated access to an SSD using MQSim
- E2E simulated access using NeSt

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

We rst compare the local access to the SSD with MQSim's performance. We notice that both local and MQSim performance are nearly identical for all the queue depths considered, with the latency observed ranging from 40-90µs. This is due to MQSim's accurate and comprehensive replication of the SSD frontend and backend, which includes details

Figure 5.5: Evaluation of a throughput sensitive read only workload

of the NAND architecture present in an SSD along with queuing at different levels. This further makes it an ideal candidate for use in NeSt as the simulated target device. We now compare NeSt's performance with real E2E storage access. In this case, NeSt's architecture resembles a host application requesting data from a storage server with a single SSD (with the same parameters) via an intermediate switch. All links are 100Gbps and the transport protocol used for NVMe-oF is also TCP. We notice that NeSt's performance is also similar to the behavior exhibited by the real experiment with the latency ranging from 120s-190 The slight difference in latency is due to the lack of kernel involvement, thus this latency is not modeled by the NeSt. Thus we can say that NeSt accurately models a real end-to-end scenario. It is to be noted that none of the scenarios are congestion scenarios, thus the carried throughput is not of interest to us. We discuss throughput evaluations for congested scenarios in subsequent sections.

5.6.2 E2E Throughput Evaluation

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

Figure 5.6: Evaluation of a throughput sensitive write only workload

Figure 5.7: Evaluation of a throughput sensitive read write workload

traf c generator. We have evaluated a read only workload, a write only workload and a read and write mixed workload. The workloads follow an exponential distribution and the end point consists of ten different storage servers. For the read only workload in Figure 5.5, the bottleneck is on the backward receive link due to the read data being transferred from the storage end exceeding the bottleneck link capacity, i.e. 100Gbps. The High, Medium and Low QoS classes read data at a rate of 90Gbps, 60Gbps and 30Gbps respectively. This causes congestion and QTCP is triggered, which we observe at the response end in Figure 5.5. The storage end also provides QoS differentiation due to the NVMe WRR protocol coupled with the proposed modi cation in 5.4.4. The three classes receive throughput differentiation in the ratio of 3:2:1. Hence we can say that E2E throughput differentiation is provided according to the application's QoS class. For the write only workload (in Figure

5.7) the bottleneck is on the sender side due to the applications' write data congesting the sender bottleneck link of 100Gbps. In this case too we notice that the QoS requirements are respected similar to the read only workload. The storage end provides differentiation too.

In the case of the read write workload, we utilized the combination of the workloads used in Figure 5.5 and Figure 5.7. This creates a pathological scenario with both sender side and response side links congested. However, as seen in Figure 5.7, NeSt shows E2E throughput differentiation even in this case. With all 3 components guaranteeing throughput requirements, i.e. request path, storage end and response path. The throughput is once again divided in the ratio of 3:2:1. This shows that NeSt's throughput differentiation capabilities (i.e. QTCP in the network end, NVMe WRR and SSD Device-level Differentiation) adequately provides E2E throughput differentiation ireespective of the workload type. We evaluate Urgent applications (i.e. applications accessing the SCM device) in subsequent sections.

5.6.3 E2E Latency Evaluation

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

Figure 5.8: Tail Latency Comparison

We now look at E2E latency evaluation in NeSt. First we test NeSt's simulation of SCM devices. In Figure 5.8a we show the access latencies exhibited by requests accessing the SCM devices. The arrival time of the requests follow a Poisson distribution while the

request sizes are xed at 4KB. This is characteristic of applications accessing SCM devices due to their need of low latency. We observe that the latency ranges from 35-40 which is surprisingly low considering that this the E2E latency of a storage request. However, our in-network prioritization using reserved buffer space in the switches helps in achieving “Urgent” class status for these applications.

In a mixed traf c scenario comprising of SCM applications and other storage applications, it is dif cult to meet latency bounds in a congestion scenario if both types of applications are latency sensitive. This is because queuing in send/receive queues can be overwhelmed, thus leading to ever-increasing latency values. Hence we consider a transient congestion episode that lasts for 2 seconds in Figure 5.8b and Figure 5.8c. The former uses DCTCP as the transport protocol and is not QoS aware while the latter uses QTCP and is QoS differentiating. The workloads for the High, Med and Low classes follow an exponential distribution with High pushing 30Gbps, Medium 60Gbps and Low 90Gbps worth of traf c. The order is reversed in this case due to the fact that these applications are latency sensitive and throughput sensitive like our evaluation in Figure 5.6.2. We observe in Figure 5.8b that DCTCP fails to provide correct QoS differentiation during the transient congestion episode, with the Medium QoS class performing far worse than the Low QoS class. However, this is remedied by using QTCP in Figure 5.8c with all four QoS classes receiving relative QoS differentiation. It is interesting to note the exact ratios of latency is not maintained as it is dif cult to control background latencies in the SSD device (caused due to the SSD's internal architecture) along with queuing in the network. However, we can still provide relative differentiation as we have shown in Figure 5.8c. Finally, we can also see that both Urgent and High classes get their differentiation in both cases. This is due to the fact that both these classes have fewer number of requests with smaller request sizes. Our evaluation in this section also shows NeSt's capability in evaluating remote applications running on different transport protocols, ex. DCTCP.

5.6.4 Testing NeSt's Trace Replay Functionality

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

Workload Characteristics

For testing NeSt's capabilities we utilized its inbuilt traf c generator to generate storage traf c following a speci c distribution. For example, in our experiments to test E2E throughput differentiation and latency differentiation we have generated four classes of traf c following a Poisson distribution, i.e. Urgent, High, Medium and Low QoS classes.

(a) (b)

Figure 5.9: Workload (a) read and (b) write distribution

For testing the storage trace replaying functionality of NeSt, we utilized the traces published by Tencent at ATC, 2020. These traces are a collection of six day long I/O traces from a production cloud block storage system. The cloud block storage system contains tens of thousands of cloud disks. We have also utilized portions of these traces to test our fat-tree architecture evaluation.

We selected traces for four different servers depending on their read and write distribution as shown in Figure 5.9a and Figure 5.9b respectively. As we can see, the Urgent class workload has the lowest read and write size counts. This is characteristic of applications

Figure 5.10: Throughput differentiation of a real workload

Figure 5.11: Fat Tree Topology

that require urgent treatmentIqbal and Chen (2023). We also chose workloads depending on their read and write mix, for example the medium class workload is write intensive while high and low class contain a good mix of both read and write requests. In a real datacenter environment, it is not expected that all vying classes of applications showcase similar behavior. We take this into consideration in our subsequent experiments.

Fat Tree and SCM Performance Evaluation

In this section we test how NeSt's SCM module performs when used as a storage device, i.e. when accessed by throughput sensitive applications. We utilize the workloads