Quality of Service for Performance-Critical Cloud Applications

by

Yiwen Zhang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2024

Doctoral Committee:

Associate Professor Mosharaf Chowdhury, Chair
Associate Professor Ryan Huang
Professor Atul Prakash
Professor Lei Ying

Yiwen Zhang
yiwenzhg@umich.edu
ORCID iD: 0000-0001-9312-9353

To Weihua Yang

ACKNOWLEDGEMENTS

The completion of a PhD journey is filled with challenges, growth, and countless moments of learning. This dissertation would not have been possible without the guidance, support, encouragement, and company of many individuals. I am deeply grateful to everyone who contributed to this dissertation, and I would like to take this opportunity to express my sincere appreciation.

First and foremost, I would like to express my deepest gratitude to my advisor, Mosharaf Chowdhury, who guided me to the world of academic research. His insightful guidance and constant encouragement convinced me to pursue a PhD degree back when I was an undergrad. I believe I was really not one of the best candidates to start a computer science PhD. In fact, the early stage of my PhD was full of failures. Without Mosharaf's patience and encouragement, I would not have been able to reach my current achievement. I am really fortunate that Mosharaf is my advisor, and I feel greatly honored to be one of his students.

I would also like to extend my heartfelt thanks to all my collaborators. My four major projects tackled problems in four different areas. Every time I entered a new area, I always received enormous help from my collaborators. I owe a special debt of gratitude to Brent Stephens, who helped me dive deep into RDMA research to finally complete my first project. I am immensely grateful to Nandita Dukkipati and Gautam Kumar, who gave me the opportunity to start my second project in datacenter networks, and guided me through with their expertise and encouragement. I am equally thankful to Ganesh Ananthanarayanan, Yuanchao Shu, and Anand Iyer, who offered me constant support in my third project on live ML analytics. I am certain what I learned from them will last long in my future career. Last but not least, I want to give my special thanks to Hasan Maruf and Jiaheng Lu, who offered enormous help in my last project on tiered memory research.

I am also sincerely thankful to Prof. Atul Prakash, Prof. Ryan Huang, and Prof. Ying Lei for serving on my thesis committee. Their valuable feedback and guidance helped me improve the quality of my thesis and defense.

My achievement would not be possible without the unwavering support of members in the Symbiotic Lab, and I express my sincere gratitude to Dr. Juncheng Gu, Dr. Peifeng Yu, Dr. Jie You, Dr. Hasan Al Muraf, Prof. Lai Fan, Jiachen Liu, Jae-Won Chung, Insu Jang, and Shiqi He. I will always miss my days at Symbiotic Lab.

Finally, my deepest, heartful thanks go to my family and my significant other for their unconditional love, support, and company. I want to express my sincere thanks to my father, Zuoqun

Zhang, and my mother, Weihua Yang, who always believe in me and do everything they can to support me throughout my life. Special thanks go to my girlfriend, Luoxi Meng. Her love guided me through the final, hardest stage of my PhD journey, and this accomplishment would not have been possible without Luoxi.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

LIST OF ALGORITHMS

LIST OF APPENDICES

ABSTRACT

Cloud infrastructure continues to scale due to rapid evolvement of both hardware and software technologies in recent years. On the one hand, recent hardware advancement such as accelerators, kernel-bypass networks, and high-speed interconnect brings more powerful computing devices, faster networking equipment, and larger data storage. On the other hand, new software technologies such as computer vision and natural language processing introduce more workloads across datacenters and the edge. As a result, more and more applications from many tenants with different performance requirements must share the compute and network resources to improve resource utilization. Therefore, it is more important than ever to ensure performance-critical applications receive the appropriate level of priority and service quality.

This dissertation aims to build system support for better quality of service (QoS) for performance-critical applications in the cloud. Specifically, we aim to provide guaranteed performance specified by service level objectives (SLOs) for multiple coexisting applications while maximizing system resource utilization. Unfortunately, we observe that existing cloud infrastructure lacks QoS support in multiple critical places including network interface cards (NICs), datacenter fabrics, edge devices and tiered memory systems, each of which requires unique QoS-aware system design to ensure predictable application performance.

To this end, we have built software solutions to provide better QoS in each of the aforementioned areas. First, we built Justitia to provide performance isolation and fairness in the NIC for kernel-bypass networks (KBNs). Justitia overcomes the unique challenges in KBN with several innovations, including split connections with message-level shaping, sender-based resource mediation with receiver-side updates, and passive latency monitoring. Second, we built Aequitas to provide QoS for latency-critical remote procedure calls (RPCs) inside datacenter networks. Aequitas is a distributed sender-driven admission control scheme that uses commodity Weighted-Fair Queuing (WFQ) to guarantee RPC-level SLOs. It enforces cluster-wide RPC latency SLOs via probabilistic downgrading in order to limit the amount of traffic admitted into different QoS levels. Third, we built Vulcan to automatically generate query plans for live ML queries based on their accuracy and end-to-end latency requirements, while minimizing resource consumption across the edge. Vulcan determines the best pipeline, placement, and query configuration by combining several techniques including Bayesian Optimization and memorizing intermediate results of pipeline operators. Fi-

nally, we built Mercury, a QoS-aware tiered memory system to provide predictable performance for memory-intensive applications. Mercury proposes a new resource management scheme inside the kernel tailored for tiered memory systems. It leverages a novel admission control and a real-time adaptation algorithm to ensure QoS guarantees for both latency-sensitive and bandwidth-intensive applications. Together, these solutions provide the missing pieces from the edge to the cloud to enable QoS for performance-critical cloud applications.

CHAPTER 1

Introduction

Due to the rapid evolution of hardware technologies, the scale of cloud infrastructure continues to grow. CPUs and GPUs are getting faster and more power efficient every year to handle the most complicated computation. Meanwhile, new breakthrough in high-speed interconnect (e.g., CXL) [15] has provided higher memory capacity with lower cost, allowing more workloads to be deployed in the cloud. Moreover, recent development of kernel-bypass networks [237, 92, 150] and 5G technologies [8, 7, 12] has brought faster networks, which has become the key enabler for compute and memory disaggregation, allowing cloud infrastructure to scale with higher resource utilization.

At the same time, recent advancement in software has brought many applications with diverse performance characteristics to the cloud. For example, as shown in Figure 1.1 in-memory key-value stores [18, 17] are widely used in user-facing microservices and they require their latency performance to be within tens of μs. Database systems leveraging persistent memory [1, 170] have less stringent latency requirement but can still achieve sub-ms performance. On the other hand, distributed storage [73], autonomous driving perception [196, 227, 228], and AI inference tasks [213, 163] perform well in the ms-scale, but they each have their own performance requirement in order to deliver proper service. On the other side of the spectrum, we have machine learning training, which can take days or even months depending on the size of the model and available compute resource.

Although modern cloud infrastructure has grown to accommodate more applications with increasing demands, applications have to share cloud resource, such as network and compute, in order to achieve high overall resource utilization in the cloud. To deploy multiple applications, cloud infrastructure has to over-subscribe resources for statistical multiplexing as it would otherwise be too expensive to provision. This makes resource contention inevitable when multiple applications surge in their demands. As a result, critical applications' performance becomes unpredictable or even unavailable when contention is high. On the contrary, low-priority applications without stringent performance requirements still compete for resource, with a chance of obtaining higher service quality than critical applications, causing *priority inversion*.

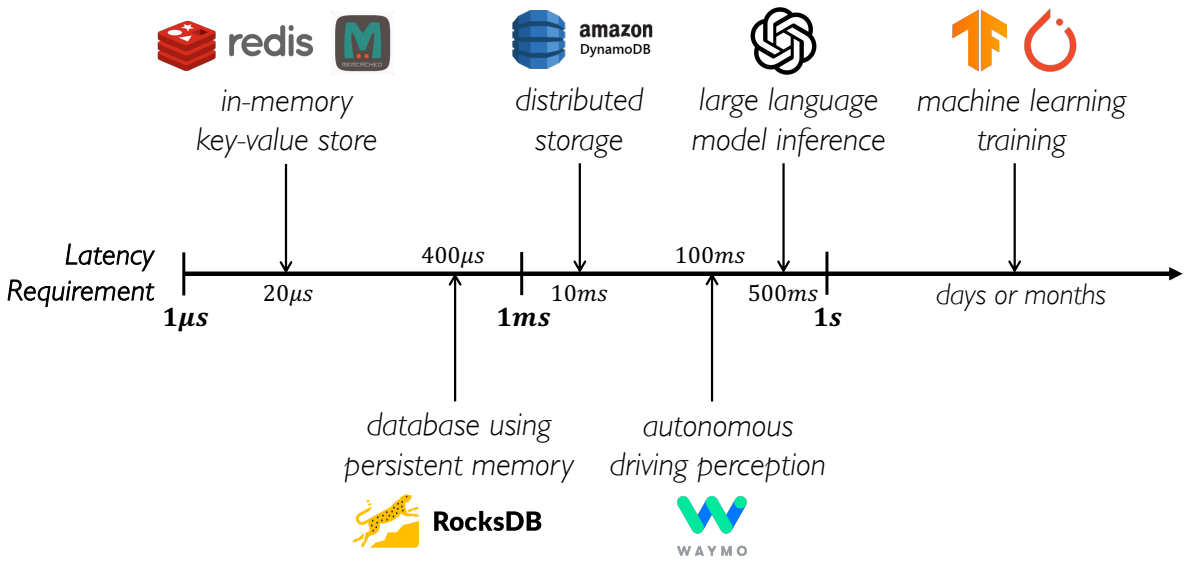


Figure 1.1: Various cloud applications with different performance characteristics.

This dissertation aims to build system support for better quality of service (QoS) for performance-critical applications in the cloud. Specifically, our goal is to provide guaranteed performance specified by service level objectives (SLOs) for multiple coexisting applications while maximize the system resource utilization. In this dissertation, we identify multiple critical places in existing cloud infrastructure that lack QoS support, including network interface cards (NICs), datacenter fabric, edge devices and tiered memory systems, each of which has unique system design challenges to ensure predictable application performance. To this end, we have built a QoS-aware system for each of the aforementioned areas, and show that it is possible to achieve both predictable performance and high resource utilization in cloud infrastructure.

We now continue this chapter with background knowledge on how existing cloud infrastructure lack QoS support, followed by the thesis statement and our research contribution.

1.1 Lack of QoS Support in Cloud Infrastructure

Existing system design in cloud infrastructure primarily focuses on improving system efficiency. The state-of-the-art systems proposed in recent research often optimize a single or one type of applications, with the goal to maximize application performance via high resource utilization. We now describe why existing systems under such a design philosophy However, such a design philosophy leads to lack of QoS support in all four research areas this dissertation focuses on:

Network interface cards. NICs are the hardware devices that optimize packet transmission performance. Hardware-based KBNs, such as Remote Direct Memory Access (RDMA), offload

packet processing tasks originally done by the operating system (OS) to specialized NICs. This allows KBNs can achieve ultra low latency and high line rate. However, such offloading takes away the operator's control over network sharing policies such as prioritization and performance isolation. As a result, multiple coexisting applications must rely on the specialized NIC to arbitrate among data transfer operations once they are posted to the hardware. Since NICs are designed to maximize utilization for optimal performance, this leads to head-of-line blocking for latency-sensitive applications, resulting in poor QoS support, as we will soon show in Chapter 2.

Datacenter fabrics. Datacenter networks rely on over-provisioning to maximize network utilization, as different applications have various peak times. However, severe network overloads can occur when multiple applications surge in their demands. During sustained overloads, critical applications suffer from latency peaks and can no longer maintain their SLO. In fact, no existing solutions can provide strong SLO guarantees for critical applications running on datacenter fabrics during high network overloads. Existing congestion control schemes [24, 238, 135, 156, 125] fair-shares network bandwidth during overloads, causing slowdowns for all application traffic. Priority-based schemes [25, 160] cannot provide QoS when flow sizes are not aligned with priority, which is indeed the case in real production environments (Chapter 3). Additionally, bandwidth sharing schemes [51, 91, 179, 34, 35, 180, 162] do not consider application priorities, nor can they provide latency guarantees for performance-critical applications.

Edge devices. Live ML analytics have gained increasing demands with large-scale deployments across edge devices [196, 227, 228, 6, 143, 10, 55]. Serving live ML queries requires ML pipeline construction, query configuration, and pipeline placement across multiple edge tiers in a heterogeneous infrastructure based on the query's accuracy and latency requirement. However, there exists no systematic approach to automatically construct pipelines based on query's end-to-end latency target. Instead, ML expert relies on past deployment experience and construct and place pipeline components manually. Such an approach overlooks the impact of pipeline construction and placement on QoS of the queries. Furthermore, recent solutions on query configurations [226, 107, 39, 198, 103], assume the ML analytics component to be a monolithic module instead of a pipeline, and thus cannot account for the additional latency and resource consumption due to different placement across the edge infrastructure.

Tiered memory systems. Tiered memory systems [151, 184, 222, 72, 194, 133] have been widely adopted to replace DRAM-only systems due to increasing memory demands of datacenter applications. Existing research on tiered memory systems primarily focuses on page temperature monitoring and efficient page migration to better utilize local memory resources (i.e., fast-tier DRAM) [184, 222, 121, 151, 20, 116]. However, these solutions optimize a single application running on a single server. They are not built with Quality-of-Service (QoS) support and thus cannot react

to applications with different service level objectives (SLOs). When multiple memory-intensive applications share the tiered memory, we observe both local memory contention and bandwidth interference. The former allows low-priority applications to grab more local memory than critical applications, whereas the latter causes significant performance degradation for latency-sensitive applications In fact, no existing solutions have systematically tackled bandwidth interference across tiered memory.

1.2 Thesis Statement and Contributions

Thesis Statement. *Existing cloud infrastructure that strives to optimize system performance often overlooks quality of service (QoS). By building QoS-aware systems in four critical areas of cloud infrastructure, we show that predictable performance and high resource utilization can be achieved simultaneously by applying a set of common design principles.*

This dissertation describes the design and implementation of the following four major systems, each of which provides QoS for performance-critical applications in one important area of the cloud software stack.

Justitia [230]. KBNs optimize application performance by offloading dataplane tasks to NICs, which lead to performance isolation anomalies when applications coexist. Justitia is the first QoS-aware system in KBNs to provide a comprehensive QoS support for multiple types of applications. Its key idea is to introduce an efficient software mediator in front of the NIC that can implement performance-related multi-tenancy policies, such as fair/weighted resource sharing and predictable latencies with maximized utilization. To overcome the unique challenges in KBN, we bring several innovations in Justitia's design. We propose the concept of split connections to decouple a tenant application's intent from its actuation. Justitia combines the benefits of sender-side and receiver-side design to proactively mediate NIC resources. It also leverages message-level shaping and passive latency monitoring to ensure latency-sensitive applications can be well isolated in front of other resource-hungry applications sharing the NIC. Justitia is shown to provide good performance isolation among several different KBN settings without losing resource utilization or incurring high CPU usage, and improve real applications' tail latency by $3.4\times$ during resource contention.

Aequitas [229]. Existing datacenter networks are over-provisioned to maximize network utilization, leading to severe network overloads that cause QoS violations. We build Aequitas, a distributed sender-driven admission control scheme to provide SLO guarantees for datacenter remote procedure calls (RPCs). Aequitas is based on two key design ideas. First, it leverage weighted fair queuing (WFQ) in commodity switches to provide delay bounds for RPCs in overload situations. Building on network calculus concepts, we derive through theory and simulations the admissible region

based on per-QoS worst-case latency with respect to QoS utilization. Second, it explicitly manages the traffic admitted on a per-QoS basis to guarantee a cluster-wide per-QoS SLO for all but the lowest QoS traffic. Specifically, Aequitas enforces traffic distribution into different network QoS levels via probabilistic downgrading in its admission control. Aequitas has been deployed in large-scale production datacenters and is able to provide SLO guarantees for critical RPCs during severe network overloads, improving RPC network latency tails by upto 5× across critical QoS levels.

Vulcan [231]. Existing ML analytics systems are not designed to serve system live ML queries, which requires deploying ML pipelines across multiple edge devices with different compute and network resource available. Vulcan is an ML analytics system we build to perform automatic query planning for live ML queries with query accuracy and end-to-end latency SLOs. It overcomes the huge search space challenge in live ML analytics by combining several key ideas. Vulcan defines a novel metric to quantify each filtering operator in order to construct the right pipeline based on a query's SLOs. It carefully identifies components of ML pipelines that are independent of placement to significantly improve the efficiency its placement search algorithm. Vulcan also propose to improve query configuration cost by borrowing ideas from Bayesian optimization. Additionally, vulcan enables fast online adaptation to ensure QoS is consistently maintained during runtime dynamics after the query is deployed to the edge. We show that Vulcan manages to provide QoS support with better performance (by up to 2.8× better) and resource efficiency (by up to 174× lower) when serving real-world live ML queries compared to state-of-the-art solutions.

Mercury. Recent research on tiered memory systems are not build with QoS support, and cannot react to applications with different SLO requirements. We introduce Mercury, the first QoS-aware tiered memory system to provide SLO guarantees for memory-intensive applications. We identify two sources of performance unpredictability – local memory contention and tier-tier memory bandwidth interference – that are unique in tiered memory. Mercury develops a new kernel-level resource management scheme to track and control memory resource for tiered memory. It combines the benefit of memory profiling and admission control to admit applications with the right amount of memory resource in order to maximize resource utilization while meeting more applications' SLO. Mercury also propose an runtime adaptation algorithm to dynamically reallocate resources during workload changes. Mercury is tested using real-world applications and shows significantly improved QoS over state-of-the-art solutions with 8.4× longer SLO satisfaction time.

1.3 Organization of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 analyzes performance isolation anomalies in KBN-specialized NICs and presents Justitia to improve QoS in KBNs. Chapter 3

introduces why modern datacenter fabrics lack QoS and how we build Aequitas to provide QoS guarantees for datacenter RPCs. Chapter 4 describes why QoS is overlooked in existing ML analytics systems and presents Vulcan to provide predictable performance for ML workloads running across edge tiers. Chapter 5 reveals source of performance unpredictability in tiered memory systems and discuss how Mercury manages to improve QoS while maximizing utilization in tiered memory. We conclude this thesis by summarizing common design principles we learned and discussing future directions in Chapter 6.

CHAPTER 2

Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks

We begin our QoS research journey with Justitia, a system we build to provide better QoS in a host machine's network interface card (NIC). Compared to traditional datacenter networking, kernel-bypass networks is more vulnerable to multi-tenancy issues because it bypasses the operating system during data transfer. In Justitia, we start with a deep analysis on isolation anomalies in RDMA, followed by a series of key ideas we leveraged to provide better multi-tenancy support without sacrificing utilization.

The remaining of this chapter is organized as follows. Chapter 2.1 gives the introduction of Justitia. Chapter 2.2 introduces the background knowledge of data transfer in hardware-based kernel-bypass networks. Chapter 2.3 describes our findings on performance isolation anomalies in RDMA. We then describe the design and implementation of Justitia in Chapter 2.4 and Chapter 2.5. Evaluation results are discussed in Chapter 2.6, followed by a discussion of related work (Chapter 2.7) and a conclusion (Chapter 2.8).

2.1 Introduction

To deal with the growing demands of ultra-low latency with high throughput (message rates) and high bandwidth in large fan-out services, ranging from parallel lookups in in-memory caches [111, 69, 113] and resource disaggregation [195, 22, 90] to analytics and machine learning [99, 19, 164], kernel-bypass networking (KBN) is becoming the new norm in modern datacenters [237, 92, 156, 62, 150]. As the name suggests, with KBN, applications bypass the operating system (OS) kernel to improve performance while relieving the CPU.

There are two major trends in KBN today. *Software-based KBN* (e.g., DPDK) removes the kernel from the data path and performs packet processing in the user space. In contrast, *hardware-based KBN* (e.g., RDMA) further lowers latency by at least one order of magnitude and reduces

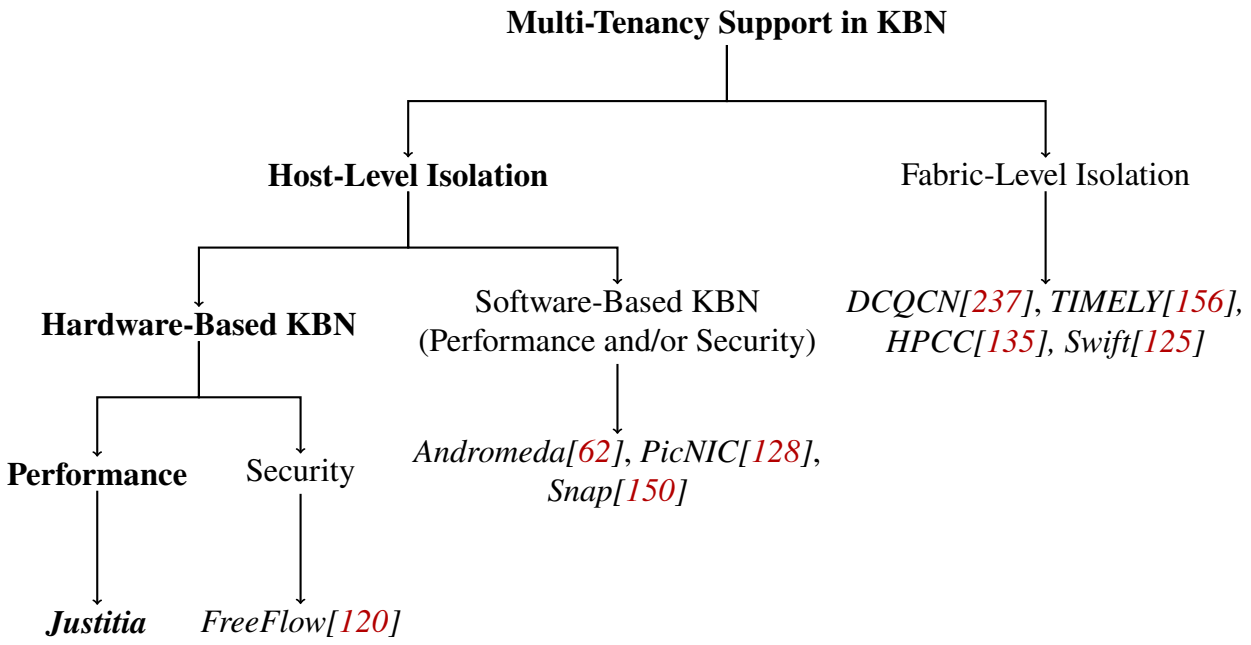


Figure 2.1: Design space for multi-tenancy support in KBN.

CPU usage by offloading dataplane tasks to specialized NICs (e.g., RDMA NICs) with on-board compute.

Hardware KBN, however, takes away the operator's control over network sharing policies such as prioritization, isolation, and performance guarantees. Unlike software KBN, coexisting applications must rely on the specialized NIC to arbitrate among data transfer operations once they are posted to the hardware. We observe that existing hardware KBNs provide poor support for multi-tenancy. For example, even for real-world applications such as DARE [178], eRPC [113], and FaSST [112], sharing the same NIC leads to severe performance anomalies including unpredictable latency, throttled throughput (i.e., lower message rates), and unfair bandwidth sharing (§2.3). In this chapter, we aim to address the following question: *Can we marry the benefits of software KBN with the efficiency of hardware KBN and enable fine-grained multi-tenancy support?*

Recent works have explored multi-tenancy support in large-scale software-based KBN deployments [62, 150, 128]. Their designs enforce fine-grained sharing policies such as performance and security (address space) isolation at the end hosts and pair with fabric-level solutions (e.g., congestion control) in case the network fabric becomes a bottleneck (Figure 2.1). However, existing software KBN solutions cannot be applied to hardware-based KBN due to three unique challenges:

1. Because host CPU is no longer involved, common CPU-based resource allocation mechanism cannot be applied. Instead, tenants issue RDMA operations with arbitrary data load at no CPU cost, which leaves no obvious point of control to exert resource mediation.
2. Hardware offloading brings packetization from user space into the NIC, disabling fine-grained

user-space shaping at the packet level [100, 190].

3. It is also crucial to preserve hardware-based KBN's efficiency (i.e., single μs latency and low CPU cost[1]) while providing multi-tenancy support.

We present Justitia, a software-only solution that enables multi-tenancy support in hardware-based KBN, to address the aforementioned challenges (§2.4). Our key idea is to introduce an efficient software mediator in front of the NIC that can implement performance-related multi-tenancy policies – including (1) fair/weighted resource sharing and (2) predictable latencies while maximizing utilization or a mix of the two. Given that RDMA is the primary hardware-based KBN implementation today, in this chapter, we specifically focus our solutions on RDMA NICs (RNICs).

Enabling fine-grained sharing policies in RDMA requires an efficient way of managing RNIC resources (i.e., link bandwidth and execution throughput). To this end, we propose *Split Connections* that decouple a tenant application's intent from its actuation and introduces a point of resource mediation. Justitia mediates RNIC resources by combining the benefits of sender-based and receiver-based design. RDMA operations are split and paced at the sender side before placing them onto the RNIC; receiver-side updates are collected to avoid spurious resource allocation caused by either incast or RDMA READ contention. Shaping is performed at the message level, where message sizes and their pacing rate are adjusted dynamically based on the current policy in use. By splitting RDMA connections, Justitia can effectively manage tenants' connections to consume RNIC resources based on the policy we set instead of letting tenants themselves compete by arbitrarily issuing RDMA operations.

To provide predictable latencies for latency-sensitive applications, Justitia introduces the concept of *reference flow* and monitors its latency instead of intercepting low-latency tenant applications. By comparing the latency measurements of many reference flows from the same sender machine to different receivers, Justitia can quickly detect (local and remote) RNIC resource contention. Given a tail latency target, Justitia maximizes RNIC resource utilization without violating the target. When the target is unachievable, based on the operator-defined policy, Justitia can choose to ensure that each of the competing n entities gets at least $\frac{1}{n}$th of one of the RNIC's two resources, extending the classic hose model of network sharing [71] to multi-resource RNICs.

We have implemented (§2.5) and evaluated (§2.6) Justitia on both InfiniBand and RoCEv2 networks. It provides multi-tenancy support among different types of applications without incurring high CPU usage (1 CPU core per host), introducing additional overheads, or modifying application codes. For example, using Justitia, DARE's tail latency improves by $3.4\times$ when running in parallel with Apache Crail [29, 209], a bandwidth-sensitive storage application, and Justitia preserves 81% of Crail's original performance. Justitia also complements RDMA congestion control protocols

[1]This does not apply to applications that aim for low latency or high message rates and busy spin their cores for maximum performance.

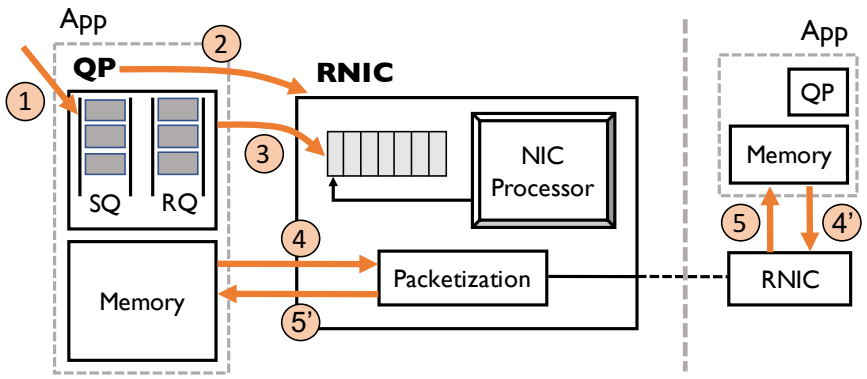


Figure 2.2: Overview of host-RNIC interaction when posting (i) an RDMA WRITE operation (① → ②. → ③ → ④ → ⑤) and (ii) an RDMA READ operation (① → ② → ③ → ④' → ⑤')).

like DCQCN [237] while further mitigating receiver-side RNIC contention, and reduces tail latency even when the network is congested.

2.2 Background

Recent works [128, 125] have discovered unpredictable latencies due to end-host resource contention, but their primary focus is on receiver-side engine congestion in software-based KBN. In this work, we aim to emphasize that *sender-side resource contention* in hardware-based KBN such as RDMA can also lead to severe performance degradation when multiple tenants coexist. An ideal solution should address both sender- and receiver-side issues. In this section, we give an overview on how an RDMA operation is performed, followed by the root cause of RDMA's lack of multi-tenancy support.

2.2.1 Life Cycle of an RDMA Operation

RDMA enables direct access between user-registered memory regions without involving the OS kernel, offloading data transfer tasks to the RNIC. Applications initiate RDMA operations by posting Work Requests (WRs) via Queue Pairs (QPs) to describe the messages to transmit. Figure 2.2 shows how an RDMA application interacts with an RNIC to initiate an RDMA operation. To start an RDMA WRITE, ① the user application place a Work Queue Element (WQE) describing the message to the Send Queue (SQ), and ② rings a door bell to notify the RNIC by writing its QP number into the corresponding doorbell register on RNIC. At this point, the user application has completed its task and offloads the rest of the work to RNIC. After the RNIC gets notified, it ③ fetches and processes the requests from the send queue, and ④ pulls the message from the user

memory, splits it into packets, and sends it to the remote RNIC. Finally, the remote RNIC ⑤ writes the received message directly into the remote memory.

In the case of an RDMA READ operation, the user application again posts the WQE and notifies the RNIC to collect it (① → ③). The local RNIC then ④' notifies the remote RNIC to pull the data from remote memory, and ⑤' places the message back to local memory after de-packetizing the received packets. Despite the opposite direction of data transfer, the remote OS remains passive just as the case with an RDMA WRITE. In both cases, the sender of the RDMA operation actively controls what goes into the RNIC while the remote side stays passively unaware.[2]

2.2.2 Lack of Multi-Tenancy Support

RDMA lacks multi-tenancy support for two primary reasons: (i) tenants/applications compete for multiple RNIC resources, and (ii) RNIC processes ready-to-consume message in a greedy fashion to maximize utilization. Both are related to different symptoms of the isolation issues.

Multi-Resource Contention There exist two primary resources that need to be shared on an RNIC: *link bandwidth* and *execution throughput*. Bandwidth-sensitive applications consume RNIC's link bandwidth to issue large DMA requests. Throughput-sensitive applications, on the other hand, consume RNIC's execution throughput to issue small DMA requests in batches. Latency-sensitive applications, however, consume neither resource with the small messages they sparsely send. As we will soon show (§2.3), isolation anomalies can occur when applications compete for different resources.

Greedy Processing for High Utilization Although the actual RNIC implementation details are private, we can consider two hypotheses on how RNIC handles multiple requests simultaneously: either the RNIC buffers WQEs collected in ③ in Figure 2.2 from multiple applications and arbitrates among them using some scheduling mechanism; or it processes them in a greedy manner. When a latency-sensitive application competes with a bandwidth-sensitive application, too much arbitration in the former can cause low resource utilization (e.g., unable to catch up the line rate), whereas too little arbitration in the latter leads to head-of-line (HOL) blocking (which leads to latency variation). Our observations across all three RDMA implementations (§2.3), where applications using small messages are consistently affected by the ones using larger ones, suggest the latter. Note that even though receiver-side congestion can also happen during step ⑤ as pointed out in [128], both root

[2]This is true even for two-sided operations that require the receiver to post WQEs to its Receive Queue before a Send Request arrives. We still consider the receiver as passive because it can only control where to place a message but cannot control *when* a message will arrive.

causes can easily stem from the sender side of the operation via step ③ and thus cannot be ignored. We elaborate on how Justitia mitigates both sender- and receiver-side issues in Section 3.4.

2.3 Performance Isolation Anomalies in RDMA

This section establishes a baseline understanding of sharing characteristics in hardware KBN and identifies common isolation anomalies across different RDMA implementations with both microbenchmarks (§2.3.1) and highly optimized, state-of-the-art RDMA-based applications (§2.3.2).

To study RDMA sharing characteristics among applications with different objectives, we consider three major types of RDMA-enabled applications:

1. *Latency-Sensitive:* Sends small messages and cares about the individual message latencies.
2. *Throughput-Sensitive:* Sends small messages in batches to maximize the number of messages sent per second.
3. *Bandwidth-Sensitive:* Sends large messages with high bandwidth requirements.

Summary of Key Findings:

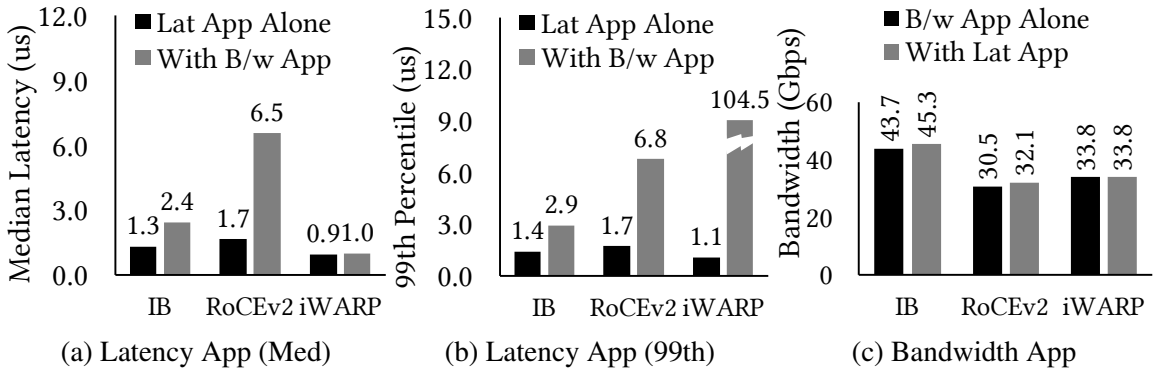
- Both latency- and throughput-sensitive applications need isolation from bandwidth-sensitive applications (§2.3.1.1).
- If only latency- or throughput-sensitive applications (or a mix of the two types) compete, they are isolated from each other (§2.3.1.2).
- Multiple bandwidth-sensitive applications can lead to unfair bandwidth allocations depending on their message sizes (§2.3.1.3).
- Highly optimized, state-of-the-art RDMA-based systems also suffer from the anomalies we discovered (§2.3.2).

In the rest of this section, we describe our experimental settings and elaborate on these findings.

2.3.1 Observations From Microbenchmarks

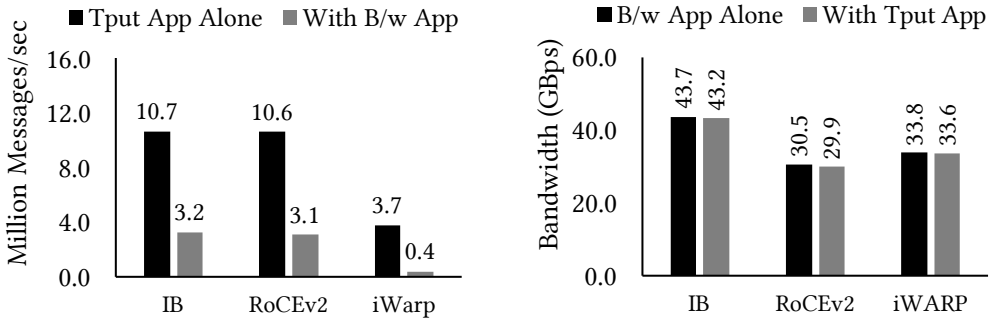
We performed microbenchmarks between two machines with the same type of RNIC, where both are connected to the same RDMA-enabled switch. For most of the experiments, we used 56 Gbps Mellanox ConnectX-3 Pro for InfiniBand, 40 Gbps Mellanox ConnectX-4 for RoCEv2, and 40 Gbps Chelsio T62100 for iWARP; 10 and 100 Gbps settings are described similar. More details on our hardware setups are in Table A.1 of Appendix A.1.

Our benchmarking applications are written based on Mellanox perftest [211] and each of them uses a single Queue Pair. Unless otherwise specified, latency-sensitive applications in our microbenchmarks send a continuous stream of 16B messages, throughput-sensitive ones send a



(a) Latency App (Med) (b) Latency App (99th) (c) Bandwidth App

Figure 2.3: Latency-sensitive applications require ioslation against bandwidth-sensitive applications.



(a) Throughput App (b) Bandwidth App

Figure 2.4: Throughput-sensitive application requires isolation from bandwidth-sensitive applications.

continuous stream of batches with each batch having 64 16B messages, and bandwidth-sensitive applications send a continuous stream of 1MB messages. Although all applications send messages using RDMA WRITEs over reliable connection (RC) QPs in the observations below, other verbs show similar anomalies as well. We defer the usage and discussion of hardware virtual lanes to Section 2.6.3.

2.3.1.1 Both Latency- and Throughput-Sensitive Applications Require Isolation

The performance of the latency-sensitive applications deteriorate for all RDMA implementations (Figure 2.3). Out of the three implementations we benchmarked, InfiniBand and RoCEv2 observes 1.85× and 3.82× degradations in median latency and 2.23× and 4× at the 99th percentile. While iWARP performs well in terms of median latency, its tail latency degrades dramatically (95×).

Throughput-sensitive applications also suffer. When a background bandwidth-sensitive appli-

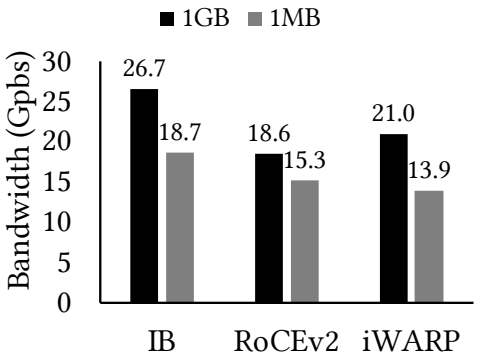


Figure 2.5: Anomalies among Bandwidth-sensitive applications with different message sizes.

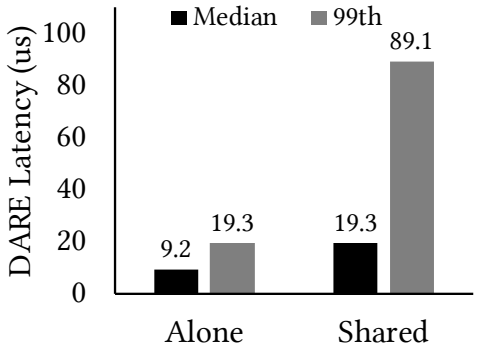


Figure 2.6: Latency of DARE's Put and Get operations when coexisting with Apache Crail's storage traffic.

cation is running, the throughput-sensitive ones observe a throughput drop of 2.85× or more across all RDMA implementations (Figure 2.4). Note that in our microbenchmark with 1 QP per application, throughput-sensitive applications that consume NIC execution throughput hit the bottleneck. This does not imply RNIC always favors link bandwidth over execution throughput. We notice RNIC bandwidth starts to become the bottleneck when there exists 4× more throughput-sensitive applications.

More importantly, both latency- and throughput-sensitive applications experience more severe performance degradations (e.g., 139X worse latency with the presence of 16 bandwidth applications) as more bandwidth-sensitive applications join the competition, which is prevalent in shared datacenters [237, 92]. Appendix A.2.1 provides more details.

2.3.1.2 Latency-Sensitive Applications Coexist Well; So Do Throughput-Sensitive Ones

We observe no obvious anomalies among latency- or throughput-sensitive applications, or a mix of the two types. Detailed results can be found in Appendix A.2.

2.3.1.3 Bandwidth-Sensitive Applications Hurt Each Other

Unlike latency- and throughput-sensitive applications, bandwidth-sensitive applications with different message sizes do affect each other.Figure 2.5 shows that a bandwidth-sensitive application using 1MB messages receive smaller share than one using 1GB messages. The latter receives 1.42×, 1.22× and 1.51× more bandwidth in InfiniBand, RoCEv2, and iWARP, respectively.

2.3.1.4 Anomalies are Present in Faster Networks Too

We performed the same benchmarks on 100 Gbps InfiniBand, only to observe that most of the aforementioned anomalies are still present. Appendix A.3.1 has the details.

2.3.2 Isolation Among Real-World Applications

In this section, we demonstrate how real RDMA-based systems fail to preserve their performance in the presence of the aforementioned anomalies.

Specifically, we performed experiments with Apache Crail [29, 209] and DARE [178]. Crail is a bandwidth-hungry distributed data storage system that utilizes RDMA. In contrast, DARE is a latency-sensitive system that provides high-performance replicated state machines through the use of a strongly consistent RDMA-based key-value store.

In these experiments, we deployed DARE in a cluster of 4 nodes with 56 Gbps Mellanox ConnectX-3 Pro NIC on InfiniBand with 64GB memory. Crail is deployed in the same cluster with one node running the namenode and one other node running the datanode.

To evaluate the performance of Crail, we launch 8 parallel writes (each to a different file) in Crail's data storage with the chunk size of the data transfer configured to be 1MB, and we measure the application-level throughput reported by Crail. To evaluate the performance of DARE, one DARE client running on the same server as the namenode of Crail issues PUT and GET operations (each PUT is followed by a GET) to the DARE server on the other 3 nodes with a sweep of message sizes from 8 byte to 1024 bytes, and we measure the application-level latency reported by DARE.

Figure 2.6 plots the latency of DARE's queries with and without the presence of Crail. In this experiment, we observe a $4.6\times$ increase in DARE's tail latency. Additionally, regardless of whether it is competing with DARE, Crail's total write throughput stays at 51.1 Gbps.

Besides DARE, highly-optimized RDMA-based RPC system such as FaSST [112] and eRPC [113] also suffer from isolation anomalies caused by unmanaged resource contention on RNICs. In fact, when background bandwidth-heavy traffic is present, FaSST's throughput experiences a 74% drop (Figure A.7) and eRPC's tail latency increases by $40\times$ (Figure A.8). More details can be found in Appendix A.3.2.

2.3.3 Congestion Control is not Sufficient

To demonstrate that DCQCN [237] and PFC are not sufficient to solve these anomalies, we performed the benchmarks again with PFC enabled at both the NICs and switch ports, DCQCN [237] enabled at the NICs, and ECN markings enabled on a Dell 10 Gbps Ethernet switch (S4048-ON). In these experiments, latency- and throughput-sensitive applications still suffer unpredictably (Sec-

tion 2.6.3 has detailed results). This is because DCQCN focuses on fabric-level isolation whereas the observed anomalies happen at the end host due to RNIC resource contention (§2.2.2).

2.4 Justitia

Justitia enables multi-tenancy in hardware-based KBN, with a specific focus on enabling two performance-related policies: (1) fair/weighted resource sharing, or (2) predictable latencies while maximizing utilization, or a mix of the two. Note that we restrict our focus on a *cooperative* datacenter environment in this work and defer strategyproofness [84, 83, 179] to mitigate adversarial/malicious behavior to future work.

Granularity of Control: We define a flow to be a stream of RDMA messages between two RDMA QPs. Justitia can be configured to work either at the flow granularity or at the application granularity by considering all flows between two applications as a whole.[3] In this work, by default, we set Justitia's granularity of control to be at the application level to focus on application-level performance.

2.4.1 Key Design Ideas

Justitia resolves the unique challenges of enabling multi-tenancy in hardware KBN with five key design ideas.

- *Tenant-/application-level connection management:* To prevent tenants from hogging RNIC resources by issuing arbitrarily large messages or creating a large number of active QPs at no cost, Justitia provides a tenant-level connection management scheme by adding a shim layer between tenant applications and the RNIC. Tenant operations are handled by Justitia before arriving at the RNIC.
- *Sender-based proactive resource mediation:* Justitia proactively controls RNIC resource utilization at the sender side. This is based on the observation that the sender of an RDMA operation – that decides when an operation gets initiated, how large the message is, and in which direction the message flows – has active control over every aspect of the transmission while the other side of the connection remains passive. Such sender-based control can react before the RNIC takes over and maintain isolation by directly controlling RNIC resources.
- *Dynamic receiver-side updates:* Pure sender-based approaches can sometimes lead to spurious resource allocation when multiple senders coexist but are unaware of each other. Justitia leverage

[3]Each granularity has its pros and cons when it comes to performance isolation, without any conclusive answer on the right one [159].

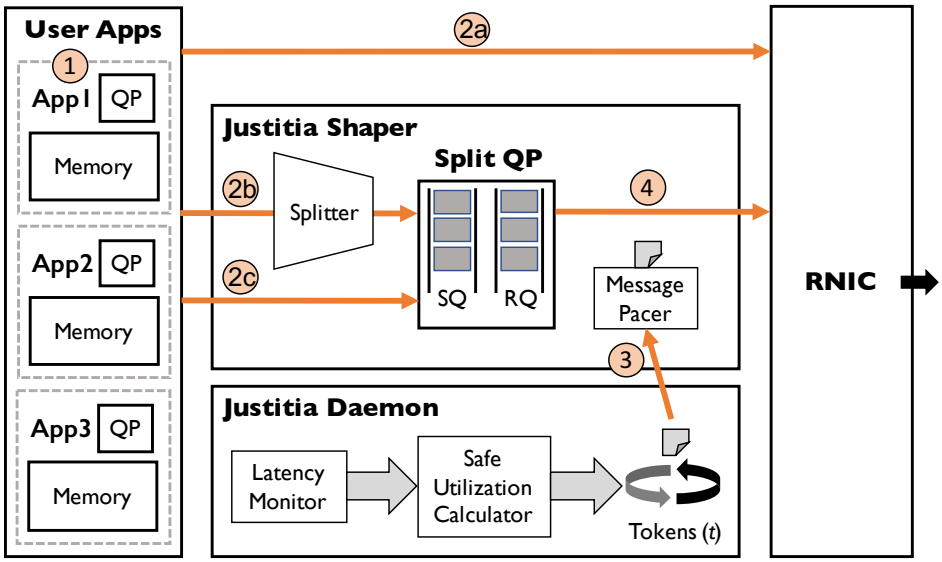


Figure 2.7: Justitia architecture. Bandwidth- and throughput-sensitive applications are shaped by tokens generated at a regular interval by Justitia. Latency-sensitive ones are not paced at all.

receiver-side updates to provide information (e.g., the arrival or departure of an application) back to the senders to react correctly when a change in the setting happens.

- *Passive latency monitoring:* Instead of actively measuring each application's latency, which can introduce high overhead, Justitia uses passive latency monitoring by issuing *reference flows* to detect RNIC resource contention.
- *Message-level shaping with splitting:* Justitia performs shaping at the message level to suit RDMA's message-oriented transport layer. At the message level, it is easy to apply specific strategies to control how messages enter the RNIC based on their sizes and the resource they consume. Large messages are split into roughly equal-sized sub-messages or chunks to (i) avoid a single message requesting too many RNIC resources; (ii) facilitate network sharing policies such as fair/weighted bandwidth share; and (iii) mitigate HOL Blocking for latency-sensitive applications.

2.4.2 System Overview

Figure 2.7 presents a high-level system overview of Justitia handling an RDMA WRITE operation (to compare with Figure 2.2). Each machine has a Justitia daemon that performs latency monitoring and proactive rate management, and applications create QPs using the existing API to perform RDMA communication. Justitia relies on applications to optionally identify their application type. By default, they are treated as bandwidth-sensitive. VMs, containers, bare-metal applications, and SR-IOV are all compatible with the design of Justitia.

As before, the user application starts an RDMA WRITE operation by ① posting a WQE into the Send Queue. Latency-sensitive applications will ②a bypass Justitia and directly interact with the RNIC as shown in Figure 2.2. The other two types of applications will enter Justitia's shaper. The Splitter will ②b split the big message from a bandwidth-sensitive applications equally into sub-messages or ②c do nothing given a small message from a throughput-sensitive application. We introduce *Split Connection* – and corresponding split queue pair (Split QP) – to handle the messages passed through the Splitter. Before sending out the message, it ③ asks the daemon to fetch a token from Justitia, which is generated at a rate to maximize RNIC resource utilization consumed by resource-hungry applications. Once the token is fetched, the Split QP ④ posts a WQE for the sub-message into its SQ and rings the door bell to notify the RNIC. The RNIC then grabs the WQE from Split QP, issue a DMA read for the actual data in application's memory region, and sends the message to the remote side (arrows not shown in the figure). Steps ③ and ④ repeat until all messages in the Split QP have been processed. The implementation details of Split QP is in Section 2.5.1.

The Justitia daemon in Figure 2.7 is a background process that performs latency monitoring and proactive rate management to maximize RNIC resource utilization when latency target is met.

2.4.3 Justitia Daemon

Justitia daemon performs two major tasks: (i) proactively manages rate of all bandwidth- and throughput-sensitive applications using the hose model [71]; (ii) ensures predictable performance for latency-sensitive applications while maximizing RNIC resource usage.

2.4.3.1 Minimum Guaranteed Rate

Justitia enforces rate based on the classic hose model [71], and always maintains a minimum guaranteed rate R_{min}:

$$R_{min} = \frac{\sum w_B^i + \sum w_T^i}{\sum w_B^i + \sum w_T^i + \sum w_L^i} \times MaxRate$$

where w_X^i represents the weight of application i of type X (i.e., bandwidth-, throughput-, or latency-sensitive), and $MaxRate$ represents the maximum RNIC bandwidth or maximum RNIC throughput (both are pre-determined on a per-RNIC basis) depending on the type of the application. The idea of R_{min} is to recognize the existence of latency-sensitive applications, and provide isolation for them by *taking out their share from the RNIC resources which otherwise they cannot acquire by themselves.* In the absence of latency-sensitive applications (i.e., $\sum w_L^i = 0$), R_{min} is equivalent to $MaxRate$, and all the resource-hungry applications share the entire RNIC resources. If all applications have equal weights, and there exist B bandwidth-, T throughput-, and L latency-sensitive applications,

Algorithm 1: Maximize $SafeUtil$

```
Function OnLatencyFlowUpdate(L, Estimated_99):
    if L = 0 then                          ▷ Reset if no latency-sensitive applications
        SafeUtil = MaxRate
    else
        if Estimated_99 > Target_99 then
            SafeUtil = max(SafeUtil / 2, R_min)
        else
            SafeUtil = SafeUtil + 1
        τ = Token_Bytes / SafeUtil
```

R_{min} can be simplified as $\frac{B+T}{B+T+L} \times MaxRate$.

In the presence of a large number of latency-sensitive applications, R_{min} could be really small, essentially removing RNIC resource guarantee. To accommodate such cases, one can fix $L = 1$ no matter how many latency-sensitive applications join the system since they do not consume much of RNIC's resources. We find this setting works well in practice (§2.6.4) and make it the default option for Justitia.

With R_{min} provided, Justitia then *maximizes RNIC's safe resource utilization* (which we denote $SafeUtil$) until the performance of latency-sensitive applications crosses the target tail latency ($Target_{99}$).

2.4.3.2 Latency Monitoring via Reference Flows

Justitia does not interrupt or interact with latency-sensitive applications because (i) they cannot saturate either of the two RNIC resources, and (ii) interrupting them fails to preserve RDMA's ultra-low latency.

Instead, whenever there exists one or more latency-sensitive applications to particular receiving machine, Justitia maintains a *reference flow* to that machine which keeps sending 10B messages to the same receiver as the latency-sensitive applications in periodic intervals (by default, $RefPeriod$ = 20 μs) to estimate the 99th percentile ($Estimated_{99}$) latency for small messages. By monitoring its own reference flow, Justitia does not need to wait on latency-sensitive applications to send a large enough number of sample messages for accurate tail latency estimation. It does not add additional delay by directly probing those applications either.

Given the stream of measurements, Justitia maintains a sliding window of the most recent $RefCount$ (=10000) measurements for a reference flow estimate its tail latency.

2.4.3.3 Maximizing $SafeUtil$

Using the selected latency measurement from the reference flow(s), Justitia maximizes $SafeUtil$ based on the algorithm shown in Pseudocode **??**. To continuously update $SafeUtil$, Justitia uses a simple AIMD scheme that reacts to $Estimated_{99}$ every $RefPeriod$ interval as follows. If the estimation is above $Target_{99}$, Justitia decreases $SafeUtil$ by half; $SafeUtil$ is guaranteed to be at least R_{min}. If the estimation is below $Target_{99}$, Justitia slowly increases $SafeUtil$. Because $SafeUtil$ ranges between R_{min} to the total RNIC resources and latency-sensitive applications are highly sensitive to too high a utilization level, our conservative AIMD scheme, which drops utilization quickly to meet $Target_{99}$, works well in practice.

To determine the value of $Target_{99}$, we constructs a latency oracle that performs pair-wise latency measurement by issuing reference flows across all the nodes in the cluster when there is no other background. Microsoft applies a similar approach in [93], which is shown to work well in estimating steady-state latency in the cluster. We adopt this approach to give a good estimate of the latency target under well-isolated scenarios.

2.4.3.4 Token Generation And Distribution

Justitia uses multi-resource tokens to enforce $SafeUtil$ among the B bandwidth- and T throughput-sensitive applications in a fair or weighted-fair manner. Each token represents a fixed amount of bytes ($Token_{Bytes}$) and a fixed number of messages ($Token_{Ops}$). In other words, the size of $Token_{Bytes}$ determines the chunk size a message from bandwidth-sensitive application is split into. A token is generated every τ interval, where the value of τ depends on $SafeUtil$ as well as on the size of each token. For example, given 48 Gbps application-level bandwidth and 30 Million operations/sec on a 56 Gbps RNIC, if $Token_{Bytes}$ is set to 1MB, then we set $Token_{Ops}$ =5000 ops and τ =167 μs.

Justitia daemon continuously generates one token every τ interval and distributes it among the active resource-hungry applications in a round-robin fashion based on application weights w_X^i. When $w_X^i = 1$ for all applications, Justitia enforces traditional max-min fairness; otherwise, it enforces weighted fairness. Each application independently enforces its rate using one of the shapers described below.

2.4.4 Justitia Shapers

Justitia shapers – implemented in the RDMA driver – enforce utilization limits provided by the Justitia daemon-calculated tokens. There are two shapers in Justitia: one for bandwidth- and another for throughput-sensitive applications.

Split Connection Justitia introduces the concept of a Split Connection to provide an interface to coordinate between tenant applications and the RNIC. It consists of a message splitter and custom *Split QPs* (§2.5.1) to initiate RDMA operations for tenants. Each application's Split Connection cooperate with Justitia daemon to pace split messages transparently.

Shaping Bandwidth-Sensitive Applications. This involves two steps: *splitting* and *pacing*. For any bandwidth-sensitive application, Justitia *transparently* divides any message larger than $Token_{Bytes}$ into $Token_{Bytes}$-sized chunks to ensure that the RNIC only sees roughly equal-sized messages. Splitting messages for diverse RDMA verbs – e.g., one-sided vs. two-sided – requires careful designing (§2.5.1).

Given chunk(s) to send, the pacer requests for token(s) from the Justitia daemon by marking itself as an active application. Upon receiving a token, it transfers chunk(s) until that token is exhausted and repeats until there is nothing left to send. The application is notified of the completion of a message only after all of its split messages have been transferred.

Batch Pacing for Throughput-Sensitive Applications. These applications typically deal with (batches of) small messages. Although there is no need for message splitting, pacing individual small messages requires the daemon to generate and distribute a large number of tokens, which can be CPU-intensive. Moreover, for messages as small as 16B, such fine-grained pacing cannot preserve RDMA's high message rates.

To address this, Justitia performs *batch pacing* enabled by Justitia's multi-resource token. Each token grants an application a fixed batch size ($Token_{Ops}$) that it can send together before receiving the next token. Batch pacing on throughput-sensitive applications removes the bottleneck on token generation and distribution; it also relieves daemon CPU cost with a unified token bucket.

Mitigating Head-of-Line Blocking. One of the foremost goals of Justitia is to mitigate HOL blocking caused by the bandwidth-sensitive applications to provide predictable latencies. To achieve this goal, we need to split messages into smaller chunks and pace them at a certain rate (enforcing $SafeUtil$) with enough spacing between them to minimize the blocking. However, this simple approach creates a dilemma. On the one hand, too large a chunk may not resolve HOL Blocking. On the other hand, too small a chunk may not be able to reach $SafeUtil$. It also leads to increased CPU overhead from using a spin loop to fetch tokens generated in a very short period in which context switches are not affordable. This is a manifestation of the classic performance isolation-utilization tradeoff. We discuss how to pick the chunk size in Section 2.5.2.

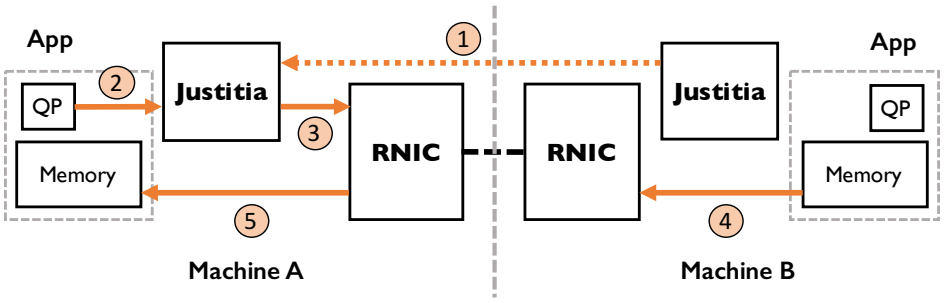


Figure 2.8: How Justitia handles READs via remote control.

2.4.5 Dynamic Receiver-Side Updates

Justitia relies on receiver-side updates to coordinate among multiple senders to avoid spurious allocation of RNIC resources. The benefits of this design is three-fold: (i) it coordinates with multiple senders to provide the correct resource allocation; (ii) it keeps track of RDMA READ issued which can collide with applications issuing RDMA WRITE in the opposite direction; (iii) it mitigates receiver-side engine congestion by rate-limiting senders with the correct fan-in information.

The updates are communicated among Justitia Daemons only when a change in the application state happened to a certain receiver (i.e., an arrival or an exit of an application) is detected. Two-sided operations, SEND and RECV, are selected in such case so that the daemon gets notified when an update arrives. Once a change is detected by a sender, it informs the receiver, which then broadcasts the change back to all the senders it connects to so that they can update the correct R_{min}. In such case, R_{min} considers remote resource-hungry application count as part of the total share. If the local daemon has not issued a reference flow and a remote latency-sensitive applications launches to the receiver, the daemon will start a new reference flow to start latency monitoring.

Handling READs RDMA specification allows remote machines to read from a local machine using the RDMA READ verb. RDMA READ operations issued by machine *A* to read data from machine *B* compete with all sending operations (e.g., RDMA WRITE) from machine *B*. Consequently, Justitia must handles remote READs as well.

In such a case, the receiver of the READ operation, machine *B*, sends the updated guaranteed utilization R_{min}, with the updated count of senders including remote READ applications) as shown in ① in Figure 2.8. After *A* receives that utilization, it operates RDMA READ by interact with Justitia normally via ② → ⑤ and enforces the updated rate.

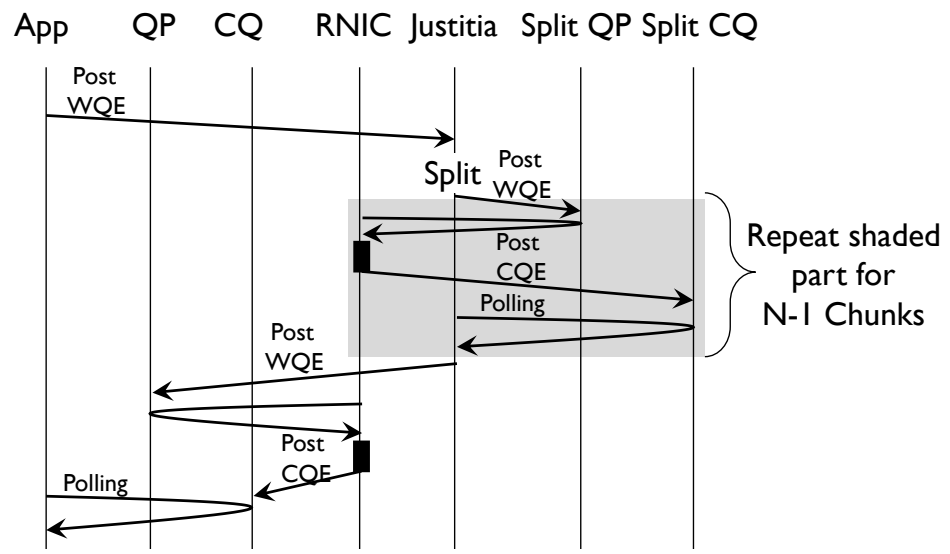


Figure 2.9: High-level overview of transparent message splitting in Justitia for one-sided verbs using Split QP. Times are not drawn to scale. Two-sided verbs involve extra bookkeeping.

2.5 Implementation

We have implemented the Justitia daemon as a user-space process in 3,100 lines of C, and the shapers are implemented inside individual RDMA drivers with 5,200 lines of C code. Our current implementation focuses on container/bare-metal applications. Justitia code is available at https://github.com/SymbioticLab/Justitia.

2.5.1 Transparently Splitting RDMA Messages

Justitia splitter transparently divides large messages of bandwidth-sensitive applications into smaller chunks for pacing. Our splitter uses a custom QP called a *Split QP* to handle message splitting, which is created when the original QP of a bandwidth-sensitive flow is created. A corresponding *Split CQ* is used to handle completion notifications. A custom completion channel is used to poll those notifications in an event-triggered fashion to preserve low CPU overhead.

To handle one-sided RDMA operations, when detecting a message larger than $Token_{Bytes}$, we divide the original message into chunks and only post the last chunk to the application's QP (Figure 2.9). The rest of the chunks are posted to the Split QP. Split QP ensures all chunks have been successfully transferred before the last chunk handled by the application's QP. The two-sided RDMA operations such as SEND are handled in a similar way, with additional flow control messages for the chunk size change and receive requests to be pre-posted at the receiver side.

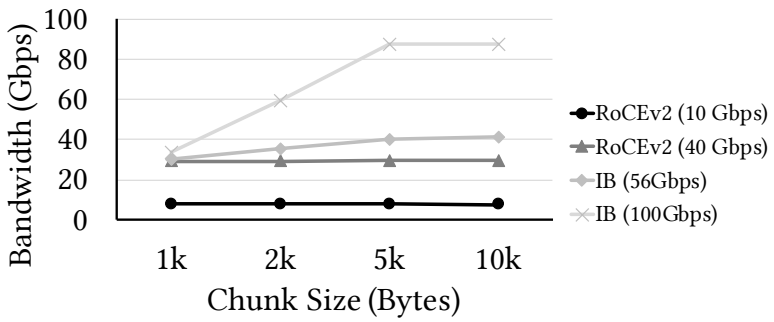


Figure 2.10: Maximum achievable bandwidth vs. chunk sizes.

2.5.2 Determining Token Size for Bandwidth Target

One of the key steps in determining $SafeUtil$ is deciding the size of each token. Because the RNIC can become throughput-bound for smaller messages instead of bandwidth-bound, we cannot use arbitrarily small messages to resolve HOL blocking. At the same time, given a utilization target, we want to use the smallest $Token_{Bytes}$ value to achieve that target to reduce HOL blocking while maximizing utilization.

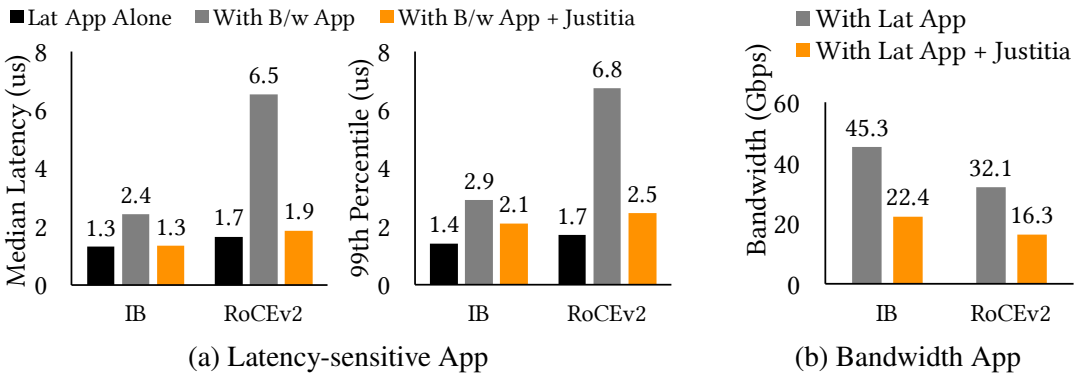
Instead of dynamically determining it using another AIMD-like process, we observe that (i) this is an RNIC-specific characteristic and (ii) the number of RNIC types is small. With that in mind, we maintain a pre-populated dictionary to store the smallest token size that can saturate a given rate (to enforce $SafeUtil$) when sending in a paced batch for different latency targets; Justitia simply uses the mappings during runtime. When latency-sensitive applications are not present, a large token size (1MB) is used. Otherwise, Justitia looks up the token size in the dictionary based on the current $SafeUtil$ value. This works well since the lower the $SafeUtil$ is, the smaller the chunk size it requires to achieve such $SafeUtil$, and the better it helps mitigating HOL blocking. Based on our microbenchmarks (Figure 2.10), we pick 5KB as the chunk size when latency-sensitive applications are present.

2.6 Evaluation

In this section, we evaluate Justitia's effectiveness in providing multi-tenancy support among latency-, throughput-, and bandwidth-sensitive applications on InfiniBand and RoCEv2. To measure latency, we perform 5 consecutive runs and present their median. We do not show error bars when they are too close to the median.

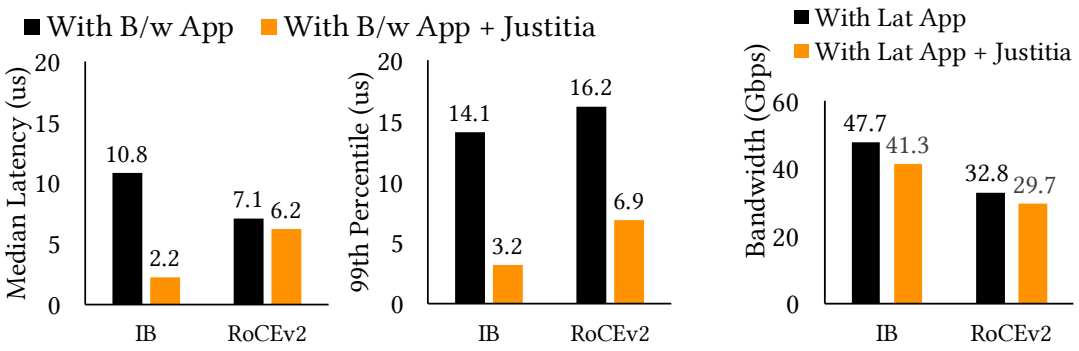
Our key findings can be summarized as follows:

- Justitia can effectively provide multi-tenancy support highlighted in Section 2.3 both in microbenchmarks and at the application-level (§2.6.1).



(a) Latency-sensitive App (b) Bandwidth App

Figure 2.11: Performance isolation of a latency-sensitive application running against a bandwidth-sensitive one.



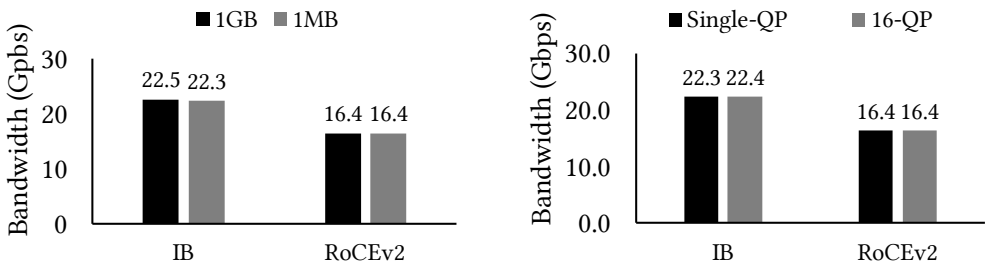
(a) Latency-sensitive App (b) Bandwidth App

Figure 2.12: Latency of a latency-sensitive application running against a bandwidth-sensitive application with 4 QPs with a relaxed latency target (10 μs).

- Justitia scales well to a large number of applications and works for a variety of settings (§2.6.2); it complements DCQCN and hardware virtual lanes (§2.6.3).
- Justitia's benefits hold with many latency- and bandwidth-sensitive applications (§2.6.4), in incast scenarios (§2.6.5), and under unexpected network congestion (§2.6.6).
  A detailed sensitivity analysis of Justitia parameters can be found in Appendix A.4.

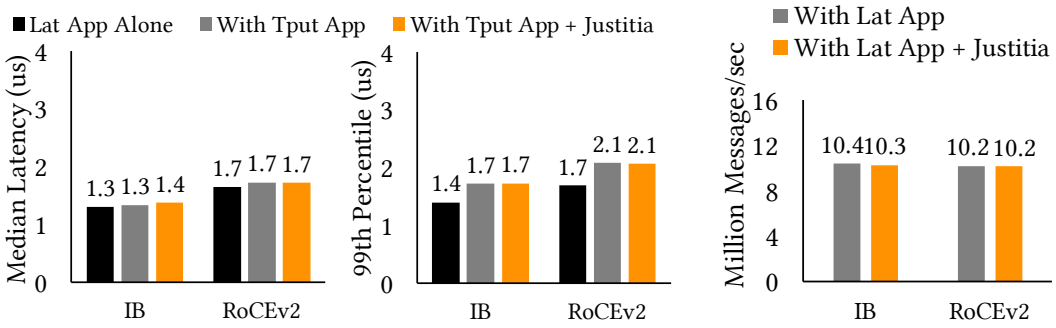
2.6.1 Providing Multi-Tenancy Support

We start by revisiting the scenarios from Section 2.3 to evaluate how Justitia enables sharing policies among different RDMA applications. We use the same setups as those in Section 2.3. Unless otherwise specified, we set $Target_{99}$ =2 μs on both InfiniBand and RoCEv2 for the latency-sensitive applications. Justitia works well in 100 Gbps networks too (Appendix A.3.1). Unless



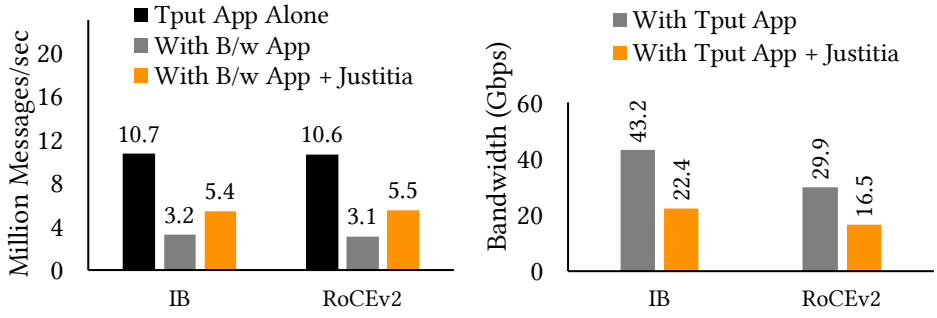
(a) Different message sizes (b) Single-QP vs. 16-QP

Figure 2.13: Fair bandwidth share of bandwidth-sensitive applications. (a) different message sizes. (b) different number of QPs.



(a) Latency-sensitive App (b) Throughput App

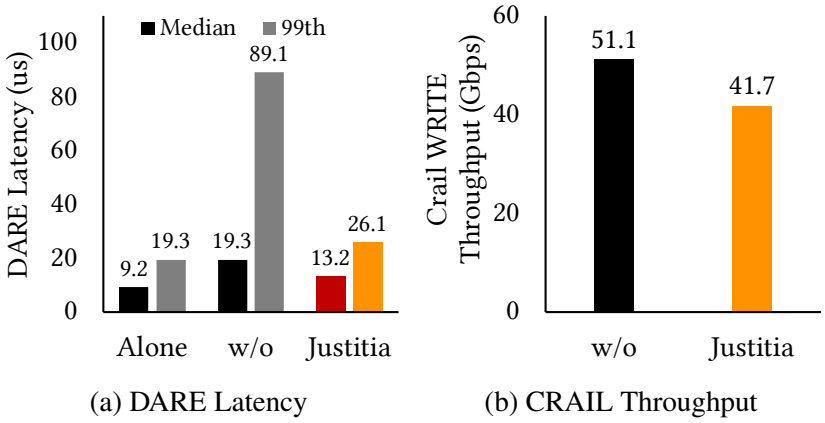
Figure 2.14: Performance isolation of a latency-sensitive application running against a throughput-sensitive application.



(a) Throughput App (b) Bandwidth App

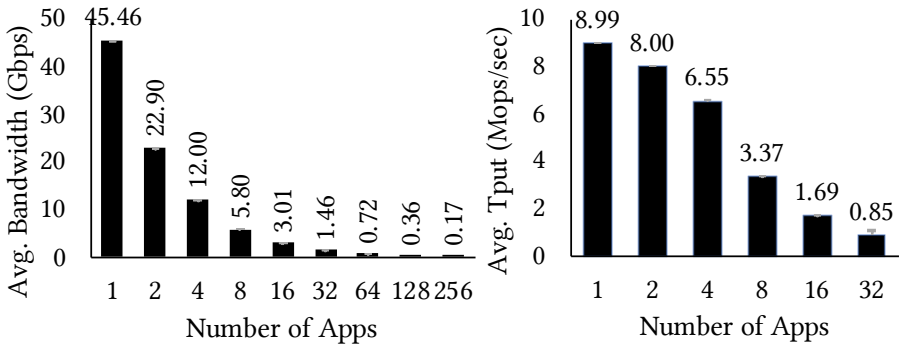
Figure 2.15: Performance isolation of a throughput-sensitive application running against a bandwidth-sensitive application.

otherwise specified, R_{min} with all applications sharing the same weights is enforced as a default policy.



(a) DARE Latency (b) CRAIL Throughput

Figure 2.16: [InfiniBand] Performance isolation of DARE running against Crail.



(a) Bandwidth-sensitive (b) Throughput-sensitive

Figure 2.17: [InfiniBand] Justitia scales to a large number of applications and still provides equal share. The error bars represent the minimum and the maximum values across all the applications.

Predictable Latency Latency-sensitive applications are affected the most when they compete with a bandwidth-sensitive application. In the presence of Justitia, both median and tail latencies improve significantly in both InfiniBand and RoCEv2 (Figure 2.11a). Due to the enforcement of R_{min}, the bandwidth-sensitive application is receiving half of the capacity (Figure 2.11b).

Next we evaluate how Justitia performs when the latency target is set to a relaxed value ($Target_{99}$ =10 μs) that can be easily met (Figure 2.12). For a slightly high $Target_{99}$, Justitia maximizes utilization, illustrating that splitting and pacing are indeed beneficial.

Fair Bandwidth and Throughput Sharing Justitia ensures that bandwidth-sensitive applications receive equal shares regardless of their message sizes and number of QPs in use (Figure 2.13) with small bandwidth overhead (less than 6% on InfiniBand and 2% on RoCEv2). The overhead becomes negligible when applying Justitia to throughput- or latency-sensitive applications (Figure 2.14).

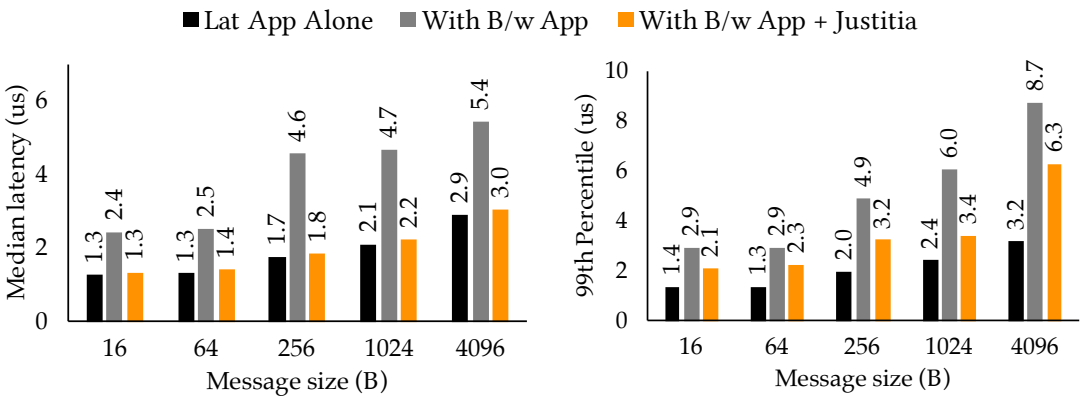


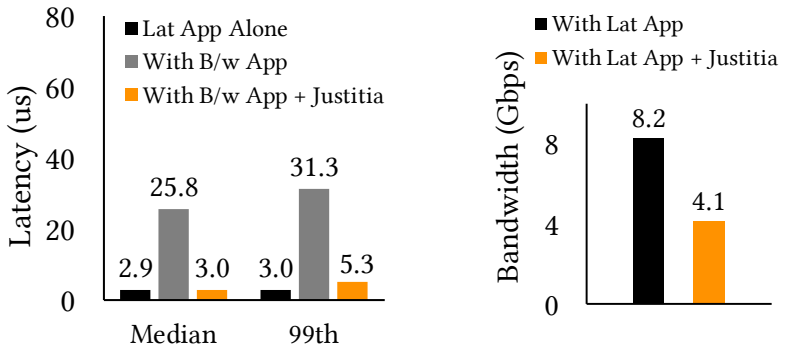
Figure 2.18: [InfiniBand] Latency-sensitive applications with different message sizes competing against a bandwidth-sensitive app.

Justitia's benefits extends to the bandwidth- vs through-sensitive application scenario as well. In this case, it ensures that both receive roughly half of their resources. Figure 2.15 illustrates this behavior. In both InfiniBand and RoCEv2, the throughput-sensitive application is able to achieve half of its original message rate of itself running alone (Figure 2.15a). The bandwidth-sensitive application, on the other hand, is limited to half its original bandwidth as expected (Figure 2.15b).

Justitia and Real-World RDMA Applications To demonstrate that Justitia can isolate highly optimized real-world applications, we performed experiments with DARE and Crail. Thanks to Justitia's high transparency, we did not need to make any source code changes in Crail (given it is bandwidth-sensitive by default), and we only changed DARE by marking it as latency-sensitive.

From these experiments, we find that Justitia improves isolation for latency-sensitive applications while also preserving high bandwidth of the background storage application. Figure 2.16 plots the performance of DARE and Crail after applying Justitia with the same setting as in Section 2.3.2. We observe that, with Justitia, DARE achieves performance that is close to running in isolation even when running alongside Crail, and Justitia improves DARE's tail latency performance by $3.4\times$ when compared to the baseline scenario while Crail also achieves 81% of its original throughput performance. This is close to the expected throughput of $\frac{8}{9}$ of Crail's original throughput since in this experiment Justitia treats the 8 parallel writes on top of Crail as separate applications.

Justitia improves performance isolation of FaSST by $2.5\times$ in throughput and eRPC by $32.2\times$ in tail latency. More details can be found in Appendix A.3.2.



(a) Latency-sensitive App (b) Bandwidth App

Figure 2.19: [DCQCN] Latency-sensitive application against a bandwidth-sensitive one.

2.6.2 Justitia Deep Dive

Scalability and Rate Conformance Figure 2.17a shows that as the number of bandwidth-sensitive applications increases, all applications receive the same amount of bandwidth using Justitia with total bandwidth close to the line rate. Justitia also ensures that all throughput-sensitive application send roughly equal number of messages (Figure 2.17b).

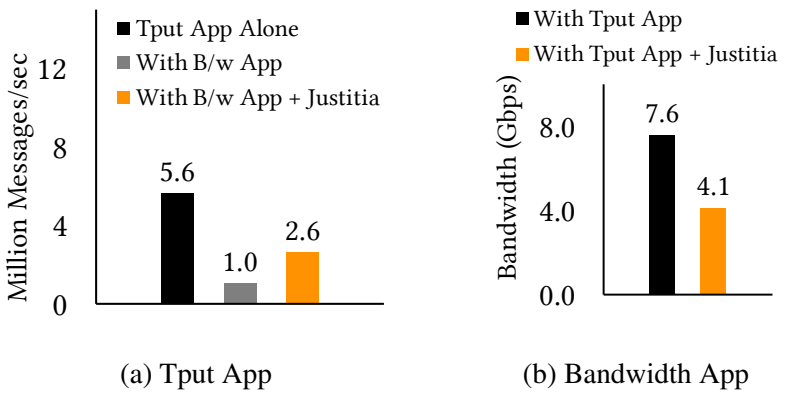
CPU and Memory Consumption Justitia daemon uses one dedicated CPU core per node to generate and distribute tokens. Its memory footprint is not significant.

Varying Message Sizes Justitia can provide isolation at a wide range of message sizes for latency-sensitive applications (Figure 2.18). The bandwidth-sensitive application receives half the bandwidth in all cases.

2.6.3 Justitia + X

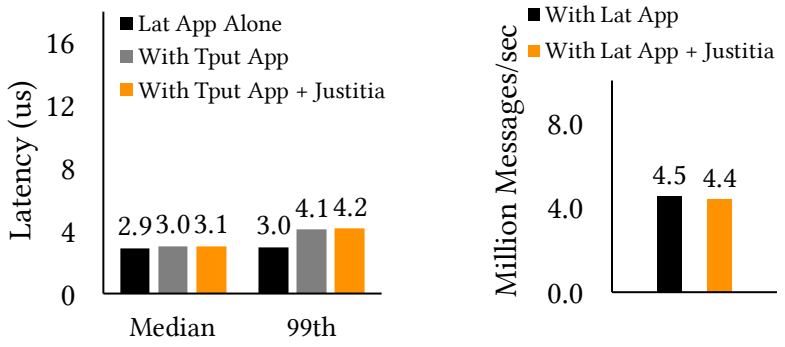
Justitia + DCQCN The anomalies we discover in this chapter does not stem from the network congestion, but rather happens at the end hosts. We found that DCQCN falls short for latency- and throughput-sensitive applications (Figures 2.19, 2.20, 2.21). Justitia can complement DCQCN and improve latencies by up to 8.6× and throughput by 2.6×.

Justitia + Hardware Virtual Lanes Although RDMA standards support up to 15 virtual lanes [31] for separating traffic classes, they only map to very few hardware shapers and/or priority queues (2 queues in our RoCE NIC) that are rarely sufficient in shared environments [126, 25]. Besides, the hardware rate limiters in the RNIC are slow when setting new rates (2 milliseconds in our setup), making it hard to use with real dynamic arrangement. Moreover, it is desirable to



(a) Tput App (b) Bandwidth App

Figure 2.20: [DCQCN] Throughput-sensitive app against a bandwidth-sensitive one.



(a) Latency-sensitive App (b) Tput App

Figure 2.21: [DCQCN] Latency-sensitive application against a throughput-sensitive one.

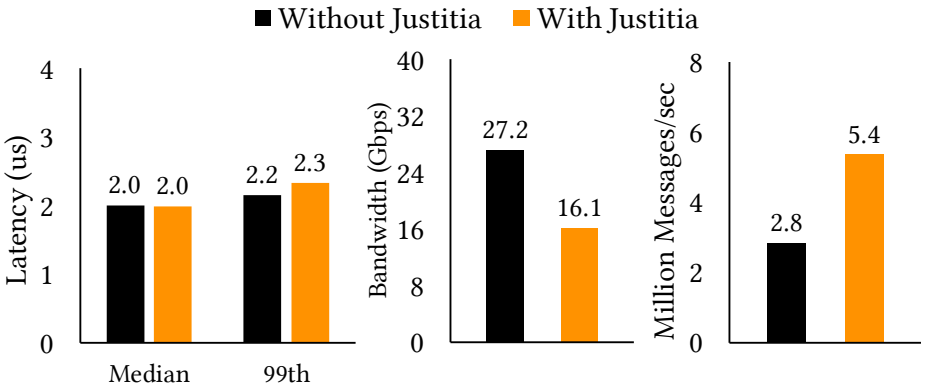
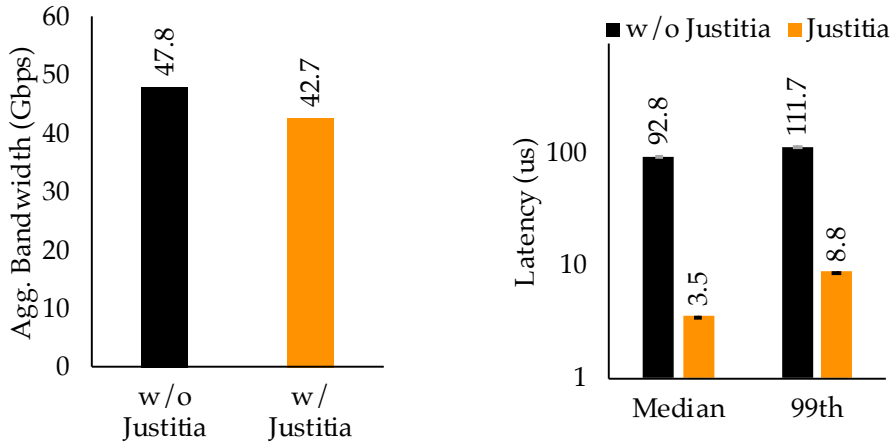


Figure 2.22: [RoCEv2] A bandwidth-, throughput-, and latency-sensitive application running on two hardware priority queues at the NIC. The latency-sensitive application uses one queue, while the other two share the other queue.

achieve isolation *within each priority queue*, as those hardware resources are often used to provide different levels of quality of service, within which many applications reside.



(a) Bandwidth Apps

(b) Latency-Sensitive Apps

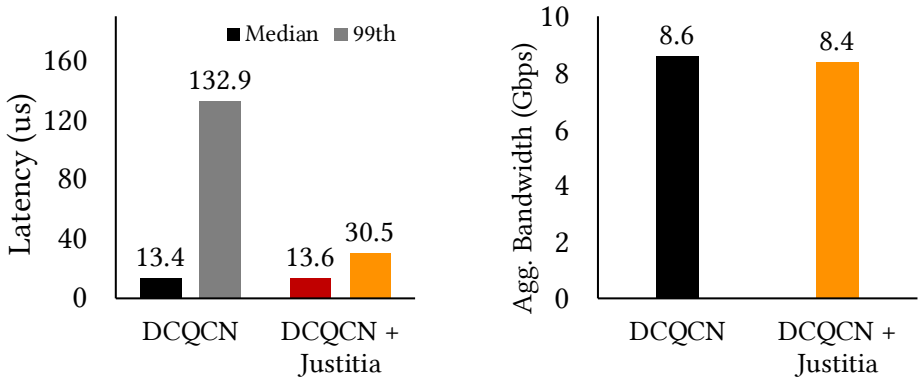
Figure 2.23: [InfiniBand] Justitia isolating 8 latency-sensitive applications from 8 bandwidth-sensitive ones. Note that 8/9th of the bandwidth share is guaranteed since Justitia counts all latency-sensitive apps as one by default (§2.4.3.3).

In this experiment, we show how limited number of hardware queues are insufficient to provide isolation and how Justitia can help in this scenario. we run three applications, one each for each of the three types (Figure 2.22). Although the latency-sensitive application remains isolated in its own class, the bandwidth- and throughput-sensitive applications compete in the same class. As a result, the latter observes throughput loss (similar to Figure 2.15). Justitia can effectively provide performance isolation between bandwidth- and throughput-sensitive applications in the shared queue.

2.6.4 Isolating among More Competitors

We focus on Justitia's effectiveness in isolating many applications with different requirements and performance characteristics. Specifically, we consider 8 bandwidth-sensitive applications – 2 each with message sizes: 1MB, 10MB, 100MB, and 1GB, and 8 latency-sensitive applications. We measure the latency and bandwidth when all the applications are active in Figure 2.23. $Target_{99}$ is set to 2μs and 20 million samples are collected for latency measurements.

Without Justitia, latency-sensitive applications suffer large performance hits: individually each application had median and 99th percentile latencies of 1.3 and 1.4 μs (Figures 2.3a and 2.3b). With bandwidth-sensitive applications, they worsen by 71.4× and 79.8×. Justitia improves median and tail latencies of latency-sensitive applications by 26.5× and 12.7× while guaranteeing R_{min} among all the applications.



(a) Latency (b) Bandwidth

Figure 2.24: [DCQCN] Incast experiment with 33 senders and a single receiver. 32 senders launch bandwidth-sensitive applications, the other sender launches a latency-sensitive application.

2.6.5 Handling Incast with Receiver-Side Updates

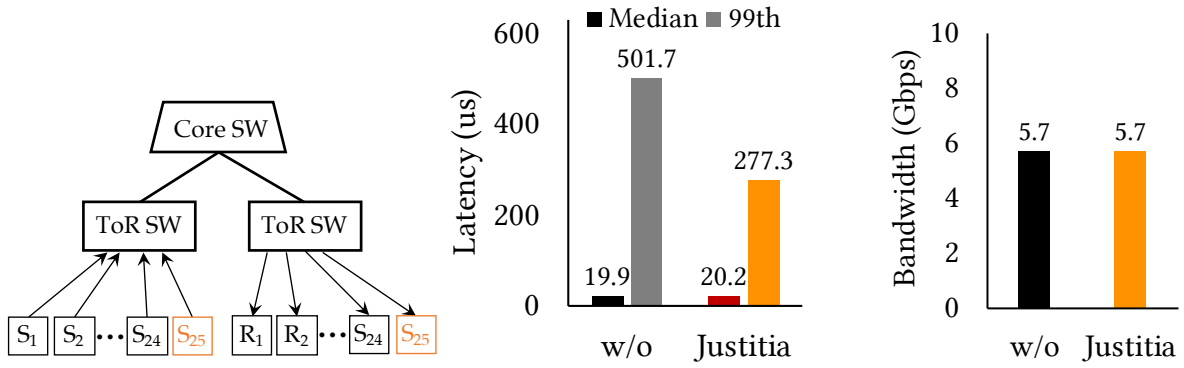
So far, we have focused on host-side RNIC contentions where the network fabric is not a bottleneck. We now evaluate how Justitia leverages receiver-side updates to handle receiver-side incast in both RoCEv2 with DCQCN and InfiniBand with its native credit-based flow control. In this experiment, 33 senders are used with the first 32 continuously launch a bandwidth-sensitive application sending 1MB messages to a single receiver. Simultaneously, the last sender launches a latency-sensitive application with messages sent to the same receiver. As described in Section 2.4.5, Justitia daemon at the receiver sends updates to all the senders whenever a sender application starts or exits, resulting in $\frac{1}{32}$-th of line rate guaranteed at each of the first 32 senders.

Figure 2.24 plots the results of this experiment, which show that Justitia still reduces tail latency even after the impact of fabric-level congestion on the reference flow latency measurements. Since the monitored latency misses the target, all the bandwidth-sensitive applications send at the minimum guaranteed rate. However, Justitia still achieves high aggregate bandwidth because this is greater than the fair share. This shows that Justitia complements congestion control and further improves the performance of latency-sensitive applications by mitigating receiver-side RNIC congestion.

We have also included a discussion on frequently asked questions regarding reference flows' impact in large-scale incast scenarios in Appendix A.5.4.

2.6.6 Justitia with Unexpected Network Congestion

When there is congestion inside the network, all traffic flowing through the network will experience increased latency, including the packets generated by Justitia as latency signals. Because today's



(a) Topology (b) Latency (c) Bandwidth

Figure 2.25: [DCQCN] Justitia's performance when Inter-ToR links are congested. Justitia achieves the same bandwidth performance because the total amount of bandwidth share on S_{25} is smaller than $SafeUtil$ due to other traffic flowing in the fabric.

switches and NICs do not report their individual contributions to end-to-end latency, Justitia cannot tell them apart. However, in practice, this is not a problem because the same response is appropriate in both scenarios.

To evaluate how Justitia performs under such cases, we performed experiments utilizing two interconnected ToR switches on CloudLab [57]. There are servers attached to each ToR switch, and every server has a line rate of 10Gbps. The experiment topology is shown in Figure 2.25a. In this topology, there is a third core switch that connects to each of the ToR switches with a link with a capacity of 160 Gbps. In this experiment, we enable DCQCN at all the servers and ECN marking at the ToR switches in the cluster.

To create a congested ToR uplink, we launch 24 bandwidth-sensitive applications each issuing 1MB messages from 24 servers (S_1–S_{24}) under one rack to the other 24 servers (R_1–R_{24}) under another rack, and none of the servers run Justitia. At the same time, we issue 8 bandwidth-sensitive applications and 1 latency-sensitive application between a pair of servers (S_{25} and R_{25}) that is controlled by Justitia. Figure 2.25 shows the performance with and without Justitia applied. Even in the case where fabric congestion is out of Justitia's control, we see that Justitia can still function correctly, and Justitia still provides additional performance isolation benefits when compared with just using congestion control (DCQCN).

2.7 Related Work

RDMA Sharing Recently, large-scale RDMA deployment over RoCEv2 have received wide attention [237, 92, 156, 157, 135]. However, the resulting RDMA congestion control algo-

rithms [156, 237, 132, 135] primarily deal with Priority-based Flow Control (PFC) to provide fair sharing between bandwidth-sensitive applications inside the network. In contrast, Justitia focuses on RNIC isolation and complements them (§2.6.3).

Justitia is complementary to FreeFlow [120] as well. FreeFlow enables *untrusted* containers to securely preserve the performance benefits of RDMA. Because it does not change how verbs are sent to queue pairs, it can still suffer from the performance isolation problems Justitia addresses. Justitia can complement FreeFlow to provide performance isolation by implementing Justitia splitter in FreeFlow's network library and Justitia daemon in its virtual router.

SR-IOV [203] is a hardware-based I/O virtualization technique that allows multiple VMs to access the same PCIe device on the host machine. Justitia design does not interfere with SR-IOV and will still work on top of it. To provide multi-tenant fairness, Justitia can be modified to distribute credits among VMs via shared memory channel similar to [120].

LITE [214] also addresses resource sharing and isolation issues in RNICs. However, LITE does not perform well in the absence of hardware virtual lanes (Appendix A.3.4).

PicNIC [128] tries to provide performance isolation at the receiver-side engine congestion in software-based kernel-bypass networks, where it utilizes user-level packet processing instead of offloading packetization to an RNIC. Hence, PicNIC's CPU-based resource allocation and packet-level shaping cannot be applied to RDMA.

Swift [125] also considers receiver-side enginer congestion in software KBN by using a dedicated enging congestion window in the congestion algorithm. However, both Swift and PicNIC ignores sender-side congestion.

Offloading with SmartNICs Recent research in SmartNICs has focused on providing programmability and efficiency in hardware offloading [117, 131, 77, 30, 140]. However, on-NIC packet orchestration leads to tens of microsecond overhead [77, 206], making performance-related multi-tenancy support still an open problem.

NICA [75] provides isolation for FPGA-based SmartNICs by I/O channal virtualization and time-sharing of the Acceleration Functional Units. Justitia focus on normal RNICs and does not require hardware changes.

Link Sharing Max-min fairness [102, 65, 38, 199] is the well-established solution for link sharing that achieves both sharing incentive and high utilization, but it only considers bandwidth-sensitive applications. Latency-sensitive applications can rely on some form of prioritization for isolation [25, 96, 219].

Although DRFQ [83] deals with multiple resources, it considers cases where a packet sequentially accessed each resource, both link capacity and latency were significantly different than

RDMA, and the end goal is to equalize utilization instead of performance isolation. Furthermore, implementing DRFQ required hardware changes.

Both Titan [207] and Loom [205] improve performance isolation on conventional NICs by programming on-NIC packet schedulers. However, this is not sufficient for RDMA performance isolation because it schedules only the outgoing link. Further, Justitia works on existing RNICs that are opaque and do not have programmable packet schedulers.

TAS [118] accelerates TCP stack by separating the TCP fast-path from OS kernel to handle packet processing and resource enforcement. However, TAS does not solve the type of isolation anomalies Justitia deals with. Justitia's design idea can be applied to improve isolation for TAS.

Datacenter Network Sharing With the advent of cloud computing, the focus on link sharing has expanded to network sharing between multiple tenants [159, 179, 51, 34, 197, 28]. Almost all of them – except for static allocation – deal with bandwidth isolation and ignore predicted latency on latency-sensitive applications.

Silo [104] deals with datacenter-scale challenges in providing latency and bandwidth guarantees with burst allowances on Ethernet networks. In contrast, we focus on isolation anomalies in multi-resource RNICs between latency-, bandwidth-, and throughput-sensitive applications.

2.8 Concluding Remarks

We have demonstrated that RDMA's hardware-based kernel bypass mechanism has resulted in lack of multi-tenancy support, which leads to performance isolation anomalies among bandwidth-, throughput-, and latency-sensitive RDMA applications across InfiniBand, RoCEv2, and iWARP and in 10, 40, 56, and 100 Gbps networks. We presented Justitia, which uses a combination of sender-based resource mediation with receiver-side updates, Split Connection with message-level shaping, and passive machine-level latency monitoring, together with a tail latency target as a single knob to provide network sharing policies for RDMA-enabled networks.

CHAPTER 3

Aequitas: Admission Control for Performance-Critical RPCs in Datacenters

Following Justitia, we extend our QoS research from a host machine into datacenter networks. In this chapter, we discuss potential QoS issues in network overloads, which is common and inevitable in datacenters, as long as multiple multiple applications are deployed and have their peak times. To provide better QoS in case of network overloads, we build Aequitas, whose core is an admission control algorithm. Aequitas works on the RPC level, and provides SLO guarantees on the network component of RPC latency by leveraing Weighted-Fair Queueing (WFQ).

The remaining of this chapter is organized as follows. Chapter 3.1 gives the introduction of Aequitas. Chapter 3.2 describes the background and motivation of datacenter RPCs and why they need better QoS support. We then take a deep dive into the theoretical analysis of WFQ delay bound, based on which we design Aequitas admission control. Chapter 3.3 and Chapter 3.5 describe the system overview and design details of Aequias. Evaluation results are discussed in Chapter 3.6, followed by a discussion of related work in Chapter 3.7. We finally conclude Aequitas in Chapter 3.8.

3.1 Introduction

Modern datacenter applications are composed of many microservices [173, 192, 152, 56] that interact with each other and remote disaggregated storage [215, 28, 212, 235, 233] using Remote Procedure Calls (RPCs). As of 2021, RPCs generate 95%+ of the application traffic in Google production datacenters, of which ~75% is to and from storage systems. To satisfy diverse business requirements of modern applications, RPCs generated by these microservices often have Service Level Objectives (SLOs) that vary widely. While many performance-critical (PC) RPCs have microsecond-scale SLOs (e.g., interactive user-facing traffic), some can take much longer (e.g., maps traffic for ride-sharing applications). Non-critical (NC) RPCs often constitute bulk storage operations, and there are best-effort (BE) RPCs from background analytics and machine learning.

Because of many recent advances in host and cluster networking, most RPCs complete within their SLOs. However, supporting predictable RPC performance under overload scenarios still remains elusive. Datacenter networks are deliberately over-subscribed for statistical multiplexing as it would otherwise be too expensive to provision. Consequently, network overloads—sometimes as high as 8× the average—are inevitable when multiple applications simultaneously surge in their demands. Under sustained network overload, the network component of RPC latency often dominates to the point of making the service effectively unavailable.

Recent research on meeting low-latency application needs in datacenter networks falls into three broad categories. First, many congestion control (CC) schemes [24, 238, 135, 156, 125] perform well in maximally using link capacity, keeping losses and network queues low in times of overload. Yet, CC by itself cannot provide *guarantees* for RPC latency: under overload, CC fair-shares the network bandwidth and causes a slowdown for *all* RPCs. Second, priority-based schemes [25, 160] minimize the average flow completion time by prioritizing smaller flows based on size or strictly apply application-defined priorities. The former does not work well when size and priority are not aligned, while the latter incentivizes applications to mark all their RPC as the highest priority. We observe both trends in production, and both lead to missed SLOs. On top of that, priority-based schemes are not readily deployable in many existing datacenters. Finally, another line of work focuses on providing bandwidth sharing guarantees [51, 91, 179, 34, 35, 180, 162], but these efforts do not consider application priorities or provide RPC latency guarantees, make restrictive assumptions on where overloads occur [197, 106] or involve centralized entities that are hard to scale in large datacenters [177].

Our goal in this chapter is to provide SLOs for *PC* RPCs—regardless of their size—in the network, even at the 99.9th percentile (99.9^{th}-p). We do so by focusing on the network component of RPC latency, which we call *RPC Network-Latency* (RNL). This leads to a design where RPCs become first-class citizens, and hosts make dynamic and local QoS-admission decisions to meet SLOs, leveraging commodity network components with weighted fair queuing (WFQ) QoS capabilities.

We present Aequitas, a simple admission control system anchored in two key conceptual insights. First, WFQ in switches has delay bounds that can be used to provide RNL SLOs in overload situations. Building on network calculus concepts [58], we derive through theory and simulations the *admissible region* based on per-QoS worst-case latency with respect to QoS utilization. Second, by explicitly managing the traffic admitted on a per-QoS basis, we can guarantee a cluster-wide per-QoS RNL SLO for all but the lowest QoS even under traffic overloads. Based on these insights, the design of Aequitas can be summarized as follows:

(1) End-hosts *align* 1:1 the priority classes at the granularity of *RPCs* (*PC*, *NC* and *BE*) to high/medium/low-weight network QoS queues (QoS_h, QoS_m, QoS_l) by encoding the QoS in the packet's DSCP header field. Switches are simple and enforce the standard QoS using WFQ.

(2) Sending hosts employ a distributed *admission control* scheme to manage the traffic mix across QoS levels. Hosts independently measure RNL for each QoS level. When the offered load of QoS_h or QoS_m RPCs is high, hosts adaptively *downgrade* excess traffic to QoS_l such that admitted traffic in higher QoS classes meets SLOs, with no explicit coordination.

We evaluate Aequitas with packet-level simulations and testbed experiments using real application workloads, and we also present early results from production deployment. We find that:

(1) Predictable latency performance can be realized cluster-wide through picking RPC winners and losers explicitly. By measuring RNL for each QoS level and realizing explicitly when the offered load is no longer in profile, hosts can make *local* decisions to admit or downgrade QoS for an RPC—a simple and effective way to ensure that quality network experience is always available for admitted traffic.

(2) With Aequitas, *PC* traffic is SLO-compliant not just at the mean, but also at the 99.9^{th}-p RNL, even when network demand spikes 10× beyond provisioned capacity. In production, Aequitas achieves 10% average reduction in 99^{th}-p RNL across fifty clusters.

(3) There exists a trade-off between how strict the SLO is and amount of traffic which can be admitted at that SLO. Aequitas achieves close to maximal traffic that can be admitted within SLO-compliance.

(4) Judicious management of traffic mix across QoS levels can create lower latency for *all* classes of traffic, including the *BE* class.

This work does not raise any ethical issues.

3.2 Background and Motivation

We begin with background on RPC workloads in modern datacenters, followed by how network impacts RPCs and what challenges exist in mitigating the network impact.

3.2.1 RPC Workloads in Modern Datacenters

Modern datacenter applications that use microservice architectures [173, 192, 152, 56] or interact with disaggregated memory and storage systems [215, 28, 212, 235, 233] rely heavily on RPCs. An RPC is a programmatic request for action or information between components of applications, and it can consist of dozens of individual packets. Hundreds of RPCs can be on the critical path to completing an application-level operation. As a result, many datacenter application developers today measure and reason about application performance in terms of RPC performance [201]. Indeed, RPCs generate 95%+ of the application traffic in our production datacenters, of which ~75% is to and from storage devices.

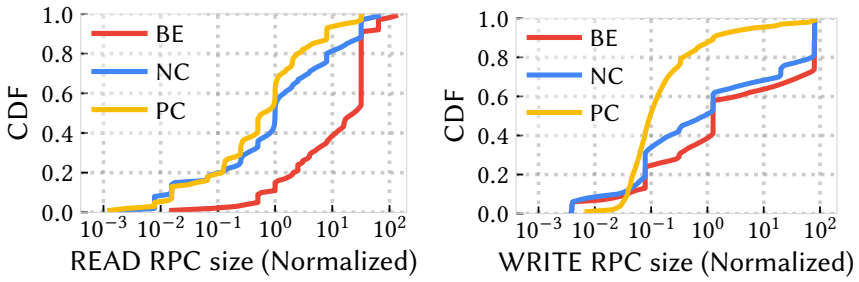


Figure 3.1: Normalized RPC size distribution of READs and WRITEs.

Business Priorities: Typical cluster applications in public clouds—Storage, MapReduce, distributed in-memory file system, web search indexing, query serving, and caching services being the largest few in our production datacenters—classify their traffic into three priority classes:

(1) *Performance-critical (PC)* RPCs have tail latency SLOs. Sometimes they are associated with real-time interactive applications or carry key control traffic.

(2) *Non-critical (NC)* RPCs generally care about sustained rate and their latency SLOs are less stringent on the tail relative to *PC* RPCs.

(3) *Best-effort (BE)* RPCs have the lowest priority, such as background backup traffic which sees no imminent disadvantage to elevated latency as long as it eventually completes. *BE* RPCs have no SLOs and are akin to a scavenger class.

RPC priority classes are used in application-level logic as well as for prioritization under server/client overloads. For storage, *PC* RPCs might constitute small random access reads and metadata exchange; *NC* RPCs might include large sequential reads, and *BE* might be for backups that are most concerned with long-term average throughput. For an online retail tenant running on public cloud, revenue-generating user traffic is *PC*; a ride-sharing tenant may consider real-time maps traffic to be *PC*; and a social networking tenant would classify its user-facing traffic to be *PC*. Machine learning training or analytics workloads may be *BE*. A key goal in assigning priorities is to avoid lower priority RPCs from interfering with those with higher priority.

Size Distributions: We find that RPC size is not fully correlated with priority from applications' perspectives. Figure 3.1 shows the CDF of storage RPC sizes using response payload size for READ RPCs and request payload size for WRITE RPCs collected in our datacenter for three categories — *PC*, *NC*, *BE* — as per their application-level priorities. While it is true that the *PC* RPCs are generally smaller than *NC* or *BE* RPCs, there are high-priority large *PC* RPCs. As such, size-based network prioritization schemes proposed in the existing literature [25, 96, 160] can lead to poor performance induced by priority inversion.

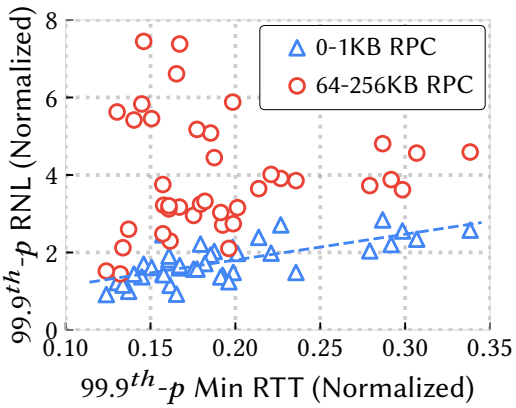


Figure 3.2: RNL of [0–1KB] and [64–256KB] RPCs versus Min RTT. Each data point is a sampled cluster.

3.2.2 Network Impact on RPCs

3.2.2.1 RPC Network Latency (RNL)

While end-to-end RPC latency serves as the primary metric for assessing regressions and triggering production alerts, it consists of two primary components: client/server latency (CPU load, thread scheduling, cache state) and network latency. In this work, we focus on meeting SLOs for the component of RPC latency impacted by network overload, which we refer to as *RPC network-latency* or RNL in this work.

Specifically, we define RNL as the time between the first RPC packet arriving at the transport layer (such as TCP) and the time when the last packet of the RPC is acknowledged at the transport. Appendix B.1 provides a detailed breakdown. RNL captures delays incurred in the host networking stack due to network overload, including queuing delays incurred due to congestion control (CC) backoff.

RNL depends on (i) the bandwidth available to an RPC, (ii) queuing and propagation delays which comprise packet-level RTT, and (iii) congestion control decisions such as congestion window and pacing rate. Figure 3.2 shows that RTT and bandwidth do not succinctly capture RNL in a randomly sampled collection of our clusters. While RNL of small (<1 MTU) RPCs correlates well with RTTs, RNL of large RPCs do not demonstrate the correlation. We can see clusters with low RNL for small RPCs (and low RTTs) but showing high RNL for large RPCs. The main reason is that an effective congestion control algorithm will keep packet delays low no matter how large the offered load is, but at high offered loads, RPCs may be queued for long periods at the sending hosts. Thus, providing either packet-level SLOs or bandwidth-SLOs in isolation are not meaningful to application developers — they care about RPC latency SLOs.

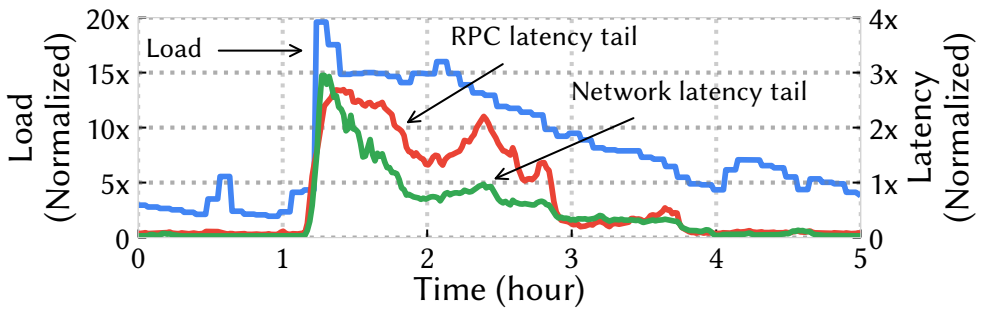


Figure 3.3: A congestion episode (w/o Aequitas) in production showing that increased load (bits/sec) leads to RPC latency spikes. Higher RNL than RPC latency in some cases is due to sampling differences.

3.2.2.2 Impact of Network Overloads on RNL

Sustained network overloads without admission control lead to queuing and packet drops, which can significantly affect RPC performance as a whole as well as RNL. Figure 3.3 shows an example of degraded RPC latency when network load surged to 8× the usual load at the ToR uplinks in a production incident. RNL is a major contributing factor to the elevated latency. For many services, such extreme degradation in latency can be tantamount to an unavailable service.

Although several proposals for network isolation [197, 106] assume that oversubscriptions occur only at the edge ToR-to-NIC links, it is not always the case. Because typical workloads use relatively little bandwidth compared to their peaks, it is cost-ineffective to provision peak capacity for *all* workloads across *all* cuts of the network. As a result, overloads can occur *anywhere* in the network along the path that an RPC takes between the client and the server. Kumar et al. [125] made a similar observation.

3.2.3 Challenges in Mitigating the Network Impact

There are three primary approaches toward enforcing performance isolation between *PC*, *NC*, and *BE* RPC traffic in the network.

(1) *Size-based approaches* — such as Shortest Job First (SJF) and Shortest Remaining Time First (SRTF) — do not work well because size and importance are often misaligned.

(2) *Strict priority queuing (SPQ)*, where RPC priorities are pushed down into the network and used as network priorities, provides a perverse incentive where developers mark all of their RPCs as *PC* to receive good network performance. More importantly, strict priority queuing is known to cause starvation if there are traffic surges with improperly configured high-priority traffic, and thus is not widely used in many production networks.

(3) *Weighted fair queuing (WFQ)* is available in commodity switches/NICs and its bandwidth

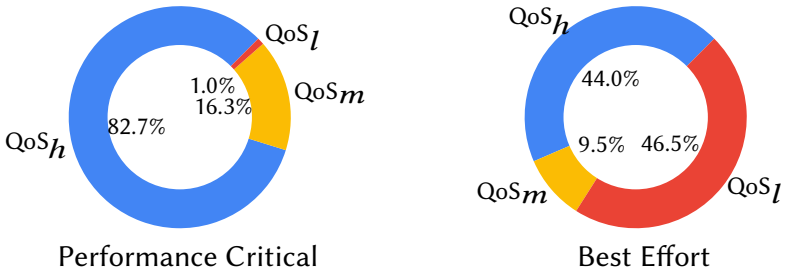


Figure 3.4: Production data showing high misalignment between RPC priority (left) and network QoS (right).

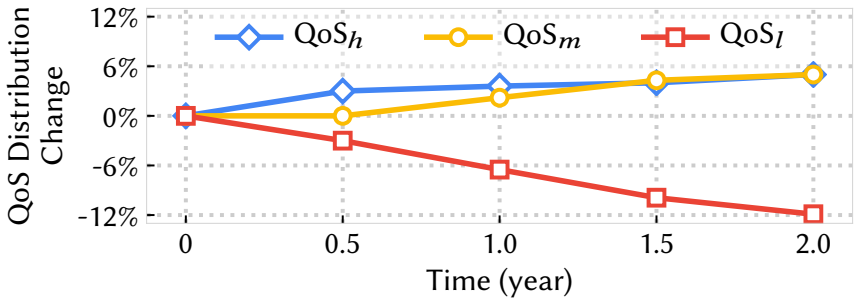


Figure 3.5: Distribution change of QoS classes over time.

sharing property enables the mapping from traffic priority classes to desired QoS levels.[1] We currently take this approach to map RPC priorities to network QoS levels[2] in our datacenters.

Mapping RPC priorities to WFQ QoS levels, however, comes with two primary constraints. First, such mappings are *coarse*; a common model used in practice by many cloud providers is to map all traffic from an entire application to the same class [200, 34, 212, 104, 28]. For example, if a business-critical application is marked as *PC*, all its traffic (including *NC* and *BE* traffic) is marked as critical. Second, QoS classes are *not associated with guarantees* or SLOs.

The common practice of allowing developers to set coarse-grained application-level priorities leads to a surprising degree of mismatch between actual RPC priority and their supposed importance in the network. In surveying our production traffic before the deployment of Aequitas, shown in Figure 3.4, we found that 17.3% of *PC* RPCs did not flow on the highest class, while 54.5% of *BE* RPCs used a higher level than necessary. Pervasive priority misalignment can degrade *PC* RPC latency due to *BE* overload even when there is sufficient capacity to meet SLO for *PC* RPCs.

A direct consequence of the above scenario is a ***race to the top***: each time a network overload

[1]We refer to WFQ [65] as the general scheduling mechanism with Virtual-Time/PGPS [176] and DWRR [199] as different implementations.

[2]NICs/switches support ~10 WFQs per port; buffer space is shared across the ports based on usage.

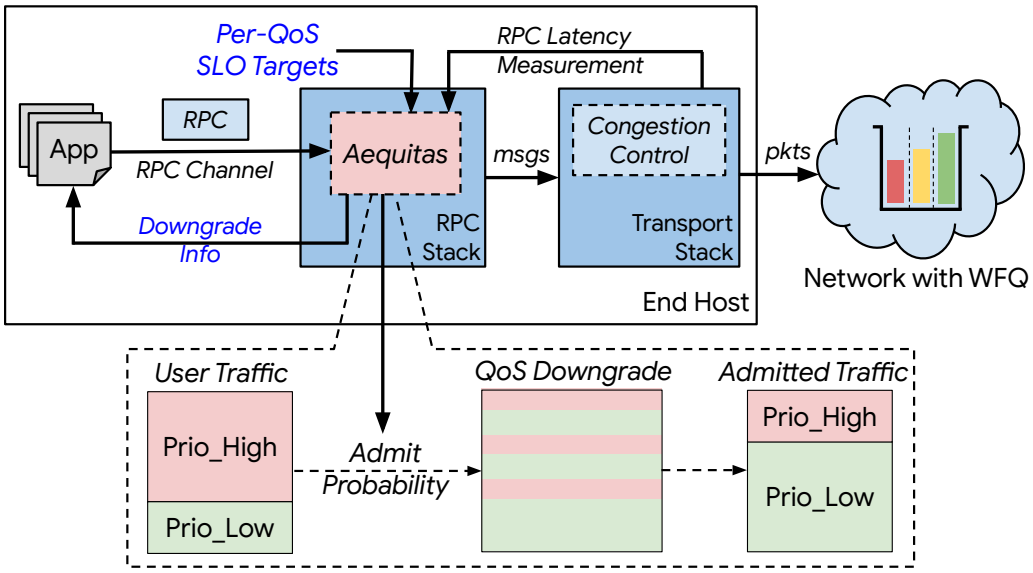


Figure 3.6: Aequitas system overview.

event occurs, applications that suffer an RNL SLO failure event are often granted a higher priority class. Figure 3.5 shows how more application traffic moved to higher classes over time for the services in our datacenters before we deployed Aequitas. Two underlying problems contribute to this problem: (1) the granularity of mapping at the application level creates stronger QoS upgrade pressure than necessary; and (2) the absence of a scheme that determines what traffic should get access to limited resources when demand exceeds network bandwidth.

3.3 Aequitas Overview

3.3.1 Objectives and Challenges

Predictable RPC completion is a key performance goal in modern datacenter networks. Our goal in this work is to *provide RNL SLOs for RPC priority classes with performance requirements (PC and NC)*. We face three challenges in achieving our goal:

(1) *Expressing SLOs for a diverse set of applications.* Latency degradation of RPCs can result from an overload of compute, storage, or networking. It is key to tease out an SLO that the network can be held accountable for.

(2) *Structural.* Given that overloads can occur anywhere in the network, the solution needs to handle dynamic overloads appearing anywhere along the path an RPC traverses.

(3) *Scale.* There may be tens of thousands of hosts, thousands of tenants, and hundreds of applications in a cluster, all requiring RPC performance at microsecond-scale.

3.3.2 System Overview

Figure 3.6 presents a high-level system diagram of Aequitas. Aequitas resides in the RPC layer and communicates with applications above it and network or transport stacks below it. Aequitas works at the RPC level and does not interfere with underlying congestion control, and it leverages weighted fair queuing (WFQ) available in commodity switches without making any modifications to existing hardware.

Applications issue RPCs on *RPC-channels*,[3] annotating their priority class, which maps to a *requested QoS* class. The operator provides the per-QoS RNL SLO targets. Once an RPC completes, Aequitas measures its RNL and feeds it into its admission control algorithm. By comparing the RNL SLO targets and the actual measurements, the algorithm adjusts the amount of traffic admitted per destination-host for the QoS at which the RPC ran. In this way, Aequitas does not need extra signaling to determine the location of oversubscription points. Admission control is enforced in a fully distributed manner among all the hosts without requiring centralized knowledge. When admitting an RPC, Aequitas adopts a probabilistic approach by maintaining *admit probability* to determine if an RPC should be admitted or *downgraded* to a lower QoS level. Downgrade information is explicitly notified back to the application as a hint to adjust their RPC priorities.

In the next section, we show theoretically why Aequitas' central idea – managing RPC traffic admitted across QoS levels with WFQ – can be a powerful knob for realizing RNL SLOs in oversubscription situations.

3.4 Analytical Results

RPC network-latency is primarily dictated by bandwidth and queuing-delay. In this section, we provide a theoretical characterization that motivated how we arrived at Aequitas design—controlling RPC network-latency across priority classes to provide differentiated SLOs by **controlling the amount of traffic admitted** on the respective QoS as realized by **WFQ**.

3.4.1 WFQ Bandwidth and Queuing-Delay Analysis

We find that WFQ is an excellent building block to help provide RNL SLOs as not only does it guarantee a *minimum bandwidth* for a traffic class, it also provides delay boundedness given its utilization level.

Given N QoS classes with $\phi_1, \phi_2, \cdots, \phi_N$ representing the weights of the WFQs that serve the

[3] An RPC-channel maps to one or more transport-layer socket/connection.

ϕ_i	QoS weight of class i
g_i	minimum guaranteed rate of class i
s_i	instantaneous service rate received for class i
r	total link capacity
a_i	instantaneous arrival rate of class i
a	aggregate instantaneous arrival rate at the link
μ	average load: average arrival rate over the period normalized to line rate
ρ	burst load: maximum instantaneous arrival rate normalized to line rate

Table 3.1: Notation used in Section 3.4 and beyond.

QoS classes, the *minimum guaranteed rate* g_i for class i with line rate r is given by $g_i = \frac{\phi_i}{\sum_j \phi_j} r$. We assume lower i indicates a higher WFQ weight. WFQ is also *work-conserving*. If the instantaneous demand for a QoS class is lower than the rate above, its traffic is completely isolated from the other QoS classes and observes nearly zero queuing delay. Correspondingly, the bandwidth share of a QoS class may exceed the rate above when other QoS classes have aggregate demands lower than their share.

The seminal work in [176] describes the *delay guarantees* supported by WFQ: 1) the delay of a QoS level can be bounded as a function of its own queue length independent of other queues and arrivals of other levels; and that 2) it is feasible to compute the worst-case queuing delay when sources are constrained by leaky-bucket rate limiters. We build upon this work using Network Calculus [58] concepts to calculate delay bounds given different utilization levels in the QoS classes instead of finding the absolute worst-case delay bounds across all possible arrival curves. Formally, if we denote the arrival rate of a class i as a_i with the sum of arrival rates as a, and define **QoS-mix** as the N-tuple $(\frac{a_1}{a}, \frac{a_2}{a}, ..., \frac{a_N}{a})$; our analysis shows *how QoS-mix affects WFQ's per-QoS delay bounds in overload situations.* We refer to the i^{th} element in the QoS-mix as **QoS$_i$-share**. Compared to prior work, the analysis is more general in that it can provide delay bounds given a QoS-mix, but is less general as the closed-form equations are restricted to only two QoS levels.

Denote x as QoS_h-share of the QoS-mix, which is the ratio of QoS_h traffic to the total arrival rate $\frac{a_h}{a}$ where $0 < x < 1$; QoS_l-share is $(1 - x)$. The ratio of QoS weights for QoS_h:QoS_l is ϕ:1. Consider the traffic pattern shown in Figure 3.7.[4] We define the entire sending period to be one unit of time. Traffic arrives in bursts characterized by a burst parameter, ρ, which is the slope of the black curve in the figure and $\rho > 1$. All notation used for analysis is defined in Table 3.1. For stability, there is an idle phase such that the average load μ within the period is less than 1.0. Thus, the delay bound can be represented as a fraction of the period; by definition, the arriving traffic can

[4]This model is similar to a Leaky-Bucket formulation expressed differently to aid closed-form equations, with the interval normalized to a unit of time.

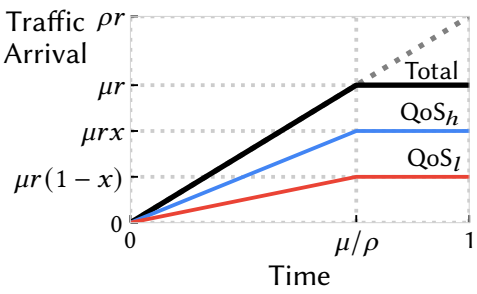


Figure 3.7: Traffic arrival pattern used in WFQ delay analysis.

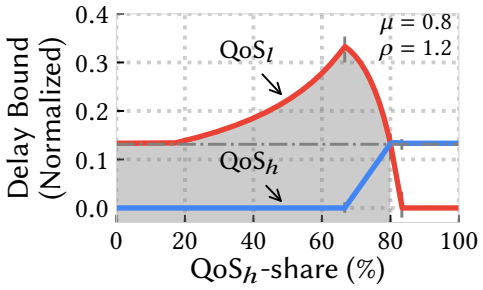


Figure 3.8: Theoretical worst-case delay with QoS_h:QoS_l weights=4:1.

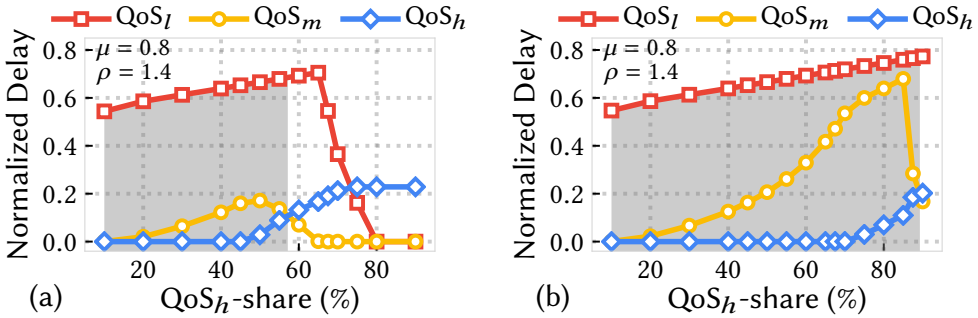


Figure 3.9: Simulated WFQ worst-case delay with 3 QoS levels under different QoS_h:QoS_m:QoS_l weights: (a) 8:4:1 and (b) 50:4:1. QoS-share of QoS_m and QoS_l is fixed at a ratio of 2:1.

be consumed within a single period. We denote this as *normalized delay bound*.

Different subdomains for x yield different service curves and thus, a different delay-bound representation, e.g., up until a certain value of x, QoS_h experiences zero delay. Also, some of the subdomains can be empty depending on the parameter values. Worst-case delay experienced by QoS_h as a function of x ($Delay_h(x)$) is given by (detailed proof is in Appendix B.2):

$$\begin{cases} 0, & x \le \frac{\phi}{\phi+1}\frac{1}{\rho} \\ \mu(\frac{\phi+1}{\phi}x - \frac{1}{\rho}), & \frac{\phi}{\phi+1}\frac{1}{\rho} < x \le \frac{\phi}{\phi+1} \\ \mu(1-x)(\phi+1-\frac{\phi}{\rho\, x}), & \frac{\phi}{\phi+1} < x \le \min\{1-\frac{1}{\phi+1}\frac{1}{\rho}, \frac{1}{\rho}\} \\ \mu(\frac{1}{\rho}-\frac{1}{\rho^2})\frac{1}{x}, & \min\{1-\frac{1}{\phi+1}\frac{1}{\rho}, \frac{1}{\rho}\} < x \le \frac{1}{\rho} \\ \mu(1-\frac{1}{\rho}), & x > \max\{\frac{\phi}{\phi+1}, \frac{1}{\rho}\} \end{cases} \tag{3.1}$$

Figure 3.8 plots the theoretical worst-case delay per QoS level in a 2-QoS scenario. There are two main takeaways from the above formulation. First, QoS-mix affects delay in both QoS classes.

As QoS distribution changes, delay from both classes experience different regions. Second, at a certain share of QoS_h, we observe *priority inversion* where delay in QoS_h exceeds that of QoS_l.

Due to the increased complexity of closed-form delay equations, we extend the analysis to more number of QoS classes via empirical analysis in simulation. Figure 3.9 plots the simulated results of a WFQ with 3 QoS levels. As before, the QoS-mix plays an important role in the delay profile among all the QoS levels. Shaded area represents delay regions with no priority inversion.

3.4.2 Controlling *QoS*-Mix for RNL SLOs

The above set of equations show that the delay bound depends on a few different parameters including QoS-mix, QoS weights, and burstiness. We characterize two lemmas to conclude that controlling QoS-mix ties directly to delay-bounds, which along with the bandwidth guarantees provided by WFQs is effective in providing RNL SLOs.

Lemma 1. *When the demand for each QoS class (a_i) exceeds its minimum guaranteed rate (g_i), the QoS-share thresholds for priority inversion are a function of QoS weights.* Priority inversion happens when the delay in a higher QoS exceeds the delay in a lower QoS. In the case where the demand for each QoS class (a_i) exceeds its minimum guaranteed rate (g_i), it takes longer to process the traffic in QoS_i versus QoS_{i+1}. The processing time is proportional to a_i/ϕ_i, i.e.,

$$a_1/\phi_1 \leq a_2/\phi_2 \leq ... \leq a_N/\phi_N. \tag{3.2}$$

For the two QoS case, this implies QoS_h and QoS_l,

$$\frac{a_h}{g_h} \leq \frac{a_l}{g_l},\ \frac{x}{\phi} \leq \frac{1-x}{1} \implies x \leq \frac{\phi}{\phi+1}$$

Thus, the values for QoS weights determine the boundary of a region of operation outside which priority inversion occurs. We define it as the *admissible region*, formally as the region where each point satisfies

$$\forall k \in \{1, 2, ..., N-1\},\ delay\_bound_k \leq delay\_bound_{k+1} \tag{3.3}$$

with N QoS classes; lower indices indicating higher priority. Figure 3.9 shows that when we increase the weight of QoS_h to 50, the priority inversion points (and hence the admissible region) move to the right, albeit at the cost of higher delay bounds for QoS_m.

Lemma 2. *While increasing QoS_h weight ϕ helps admit more QoS_h traffic with zero delay, beyond QoS_h-share exceeding $\frac{1}{\rho}$, delay is independent of QoS weights.* As ϕ increases, the domain of case 1 in Eq 3.1 grows and approaches $\frac{1}{\rho}$ as ϕ keeps increasing. In fact, based on Eq 3.1, when

ϕ goes to infinity, QoS_h delay expression becomes

$$Delay_h(x) = \begin{cases} 0, & x \leq \frac{1}{\rho} \\ \mu(x - \frac{1}{\rho}), & \frac{1}{\rho} < x \leq 1 \end{cases} \tag{3.4}$$

Taking the two lemmas together, we observe while QoS weights are an important parameter to increase the amount of traffic permitted at a given delay bound, it is *controlling the QoS-share that is effective in providing the delay bounds in the first place.* This property directly motivates a key part of Aequitas' design – control utilization at a given QoS via admission control. We also note that as ϕ increases, the equations approach the single-QoS scenario where the only way to control delay is to control the amount of admitted traffic. Furthermore, the core idea of controlling QoS-mix enables cloud operators to select SLOs from the profiles of latency versus QoS-mix.

Aligning RPC priority to network QoS with WFQ scheduling and controlling the amount of traffic admitted to individual QoS classes are the core principles of Aequitas design which we describe in the next section.

3.5 System Design

The overall design for Aequitas follows naturally from the analysis above and consists of two sequential phases amenable to incremental deployment. In this chapter, we focus on presenting the key design ideas without going into details about how Aequitas is implemented in the RPC stack and the transport stack internally at our clusters, which are beyond the scope of this chapter.

Phase 1: Align Network QoS with RPC priority Most datacenter applications have a clear notion of RPCs that are PC, NC, and BE. Classifying an entire application or a job into a single priority class is too coarse-grained; transport level flows can be both coarse and fine grained, e.g., consider the issue we face with multiplexing multiple RPCs onto a single TCP connection; packet level is needlessly fine-grained and loses application-level semantics. As a first step to providing RNL SLOs, Aequitas maps, at the *granularity of RPCs*, the three priority classes bijectively to three QoS classes served with WFQ-scheduling: PC RPCs to QoS_h, NC to QoS_m, and BE to QoS_l. Aequitas provides SLOs for QoS_h and QoS_m; QoS_l is treated as a scavenger class on which best-effort and downgraded traffic is served and offers no SLOs. The design organically extends to larger numbers of QoS priority classes.

Phase 2: Distributed Admission Control via QoS downgrade to provide RNL SLOs Aequitas uses a distributed algorithm implemented completely at sending hosts to decide, at the RPC granularity, whether to admit a given RPC on the requested QoS by controlling an *admit probability*. This controls the portion of RPCs admitted across QoS levels in order to meet RNL SLOs. In a

departure from traditional mechanisms of admission-control that either drop or rate-limit traffic, Aequitas ***downgrades*** the unadmitted RPCs and issues them at the lowest QoS level. The algorithm follows an Additive Increase Multiplicative Decrease (AIMD) control.

3.5.1 Distributed Admission Control

At its core, Aequitas is a distributed admission control system for RPCs implemented completely at sending hosts utilizing a novel mechanism of QoS-downgrade enabled by WFQs commonly available in commodity switches.

Probabilistic admission of RPCs: Central to Aequitas' distributed algorithm is an *admit probability* denoted by p_{admit} that each RPC channel maintains on a per-*(src-host, dst-host, QoS)* basis; Aequitas probabilistically admits RPCs on a given QoS based on p_{admit}, which Aequitas controls as per Algorithm 2. Note that if an RPC is downgraded, it is **explicitly notified** to the application via an additional field in RPC metadata (lines 10-11). This notification is important for two reasons: (i) the application sees network overload and QoS downgrades directly, and (ii) when not all RPCs can be admitted on the requested QoS, the application has the freedom to control which RPCs are more critical and issue only those at higher QoS to prevent downgrades. How applications exactly use the downgrade information is outside the scope of this chapter.

The key idea behind the algorithm is simple. At each source host, for each RPC channel, Aequitas collects measurements of RPC network latencies (RNL as described in §3.2.2.1) per destination host and QoS level to capture the delays incurred by both network overload and congestion control backoff. These measurements serve as the primary signal to adjust p_{admit}. If the latency is within the target, p_{admit} is increased, otherwise it is decreased. We find that such a probabilistic approach has two main advantages: (i) admit probability translates directly to determine the portion of RPCs that needs to be downgraded to control the amount of admitted traffic, and (ii) it is simple to reason about in terms of a fair and efficient distributed algorithm as we describe below. We note similarities to AQM schemes [78, 175] that also perform probabilistic admission control albeit at the packet level; Aequitas does so at the granularity of RPCs.

AIMD on admit probability: AIMD as a feedback control algorithm has been widely used to provide fair and efficient utilization of resources both in theory and in practice [101, 49]. Aequitas' usage of AIMD has several important differences compared to how other systems use AIMD.

Additive increase: Aequitas increases p_{admit} if the observed RNL is below the target, restricted to one update per $increment\_window$ (lines 15-18). The rationale is that for fairness, the increment in p_{admit} should be agnostic to how many RPCs each channel is sending. The value of $increment\_window$ depends on the percentile at which the SLO is defined, e.g., if the SLO is defined at the 99.9^{th}-p, the $increment\_window$ is higher than the case where it is at the 99^{th}-p—the

Algorithm 2: QoS Downgrade Algorithm

1 **Notation:**
α: additive increment, β: multiplicative decrement,
$target\_pctl$: target percentile of tail latency, N: total number of QoS levels, $priority$: RPC priority class specified by the application, $size$: size of an RPC in number of MTUs.

2 **Initialization:**
for $i \leftarrow 1$ **to** $N-1$ **do**
3 $p\_admit[i] = 1$
4 $increment\_window[i] = latency\_target[i] \cdot \frac{100}{100 - target\_pctl[i]}$

5 **On RPC Issue** $(rpc,\ priority)$**:**
6 $QoS_{req} \leftarrow MapPriortyToQoS(priority)$
7 **if** $rand() \leq p\_admit[QoS_{req}]$ **then**
8 $QoS_{run} \leftarrow QoS_{req}$
9 **else**
10 $QoS_{run} \leftarrow QoS_{lowest}$
11 $rpc.is\_downgraded \leftarrow True$
12 $RPC\_Start(rpc, QoS_{run})$

13 **On RPC Completion** $(rpc\_latency,\ size,\ QoS_{run})$**:**
14 $k \leftarrow QoS_{run}$
15 **if** $rpc\_latency\ /\ size < latency\_target[k]$ **then**
▷ Additive Increase
16 **if** $now - t\_last\_increase[k] > increment\_window[k]$ **then**
17 $p\_admit[k] \leftarrow \mathbf{min}(p\_admit[k] + \alpha,\ 1)$
18 $t\_last\_increase[k] \leftarrow now$
19 **else** ▷ Multiplicative Decrease
20 $p\_admit[k] \leftarrow \mathbf{max}(p\_admit[k] - \beta \cdot\ size,\ floor)$

algorithm is more conservative in increasing the admit probability when the SLO is for a higher tail.

Multiplicative decrease: If the RPC misses the specified SLO, p_{admit} is decreased by a constant amount per SLO miss (lines 19-20). Aequitas achieves fairness across RPC-channels. For this, when overload occurs, a channel sending more RPCs incurs a larger decrease in its p_{admit} versus a channel sending fewer RPCs. We utilize RPC-level clocking to achieve this: the constant decrement in the admit probabilities implies that the overall decrease in a given time interval becomes proportional to the RPCs on the channel that miss the SLO.[5] We set a threshold below which p_{admit} does not further decrease. This is to prevent starvation – when the admit probability drops to zero, no new RPCs get admitted on the requested QoS, resulting in no further latency measurement for the admit probability to grow. Detailed evaluation of how the algorithm achieves fairness and efficiency is in §3.6.5.

[5] An implication of this is if a channel's rate of RPCs is within its fair share, its admit probability will converge to 1.0.

By applying the above AIMD policy on the admit probability, Aequitas is able to control the rates of issuing RPCs in a fair and efficient way, and converge to a stable QoS-mix with which the given set of SLOs are precisely maintained. As we will show in §3.6.3, this leads to close to maximal admitted traffic while maintaining SLO-compliance.

Handling different RPC sizes: We make two augmentations to the algorithm to handle different RPC sizes. First, the $latency\_target$ is specified as a normalized SLO on an MTU basis, enabling larger RPCs to have a higher absolute RNL target. Second, the multiplicative decrease is made proportional to the size of the RPC, such that an SLO miss on, say, a 10-packet RPC behaves similarly to SLO misses on ten 1-packet RPCs. In other words, irrespective of the sizes of the RPCs, Aequitas will converge to its fair share.

3.5.2 SLO Guarantees and Robustness

The trifecta of aligning priorities, providing per-QoS SLOs (except the lowest QoS), and admission control to maintain a QoS-mix enables a systematic use of datacenter QoS and incentivizes applications to be well-behaved when using higher QoS.

A naive way of meeting SLOs is to admit a very small number of RPCs on each QoS. However, Aequitas aims to maximize the traffic that is admitted (performance-criterion) while retaining SLO-compliance (correctness-criterion). Additionally, we can show that in our *theoretical* model, at least $r\frac{\phi_i}{\sum\phi}\frac{\mu}{\rho}$ traffic is admitted in QoS_i except for the lowest QoS To see why this is true, consider the model in §3.4.1. Denote X_i as the average rate that will at least be admitted in QoS_i under Aequitas. Given X_i, the maximum instantaneous rate on QoS_i can be represented as $X_i\frac{\rho}{\mu}$. When the arrival rate does not exceed its minimum guaranteed rate, there cannot be any delay on QoS_i (Appendix B.2.1 has a formal proof on this), and all traffic is admitted. Thus, if the maximum instantaneous arrival rate for QoS_i is less than g_i, X_i is guaranteed to be admitted:

$$X_i\frac{\rho}{\mu} \leq g_i = \frac{\phi_i}{\sum\phi}r,\ X_i \leq r\frac{\phi_i}{\sum\phi}\frac{\mu}{\rho}$$

with any additional SLO resulting in a larger value of X_i. Note that the guaranteed share is inversely proportional to the traffic burstiness and we evaluate this aspect in §3.6.4.

It is important to note that while Aequitas provides latency SLOs for all *admitted* RPCs, it does not guarantee the amount of traffic admitted on a per-application or per-tenant basis–wherein the admitted traffic depends on the number of co-existing applications/tenants, as Aequitas shares the per-QoS bandwidth. One can augment Aequitas to provide application/tenant traffic rate guarantees with a centralized RPC *quota server*, and we leave this for future work.

3.6 Evaluation

Our evaluation consists of event-driven simulation, testbed experiments, and results from production deployment. The focus is on two aspects, first is to see if Aequitas remains SLO-compliant by controlling the 99^{th}-p (or 99.9^{th}-p) RNL which is our *correctness* criterion, and second is whether Aequitas admits close to ideal amount of traffic irrespective of the input traffic mix which serves as the main *performance* criterion. We evaluate these and additional aspects such as fairness and convergence over different topologies, RPC size distributions—both synthetic and from production, traffic patterns and burstiness. All results are at 100Gbps link rates.

3.6.1 Simulator

We use a packet-level simulator[6] built atop YAPS [127], augmenting it with WFQ scheduling in switch queues, Swift [125] congestion control, and an RPC stack where Aequitas is implemented. Our open source simulator also serves as a tool for datacenter operators to help define the admissible region and set the right SLOs. Unless otherwise specified, we use an α value of 0.01 and a β value of 0.01 per MTU (note the size-based adjustment to multiplicative decrease in Algorithm 2).

Validation: We validate the correctness of the simulator by replaying the theoretical 2-QoS scenario that was shown in Figure 8. Congestion control is disabled and the buffer size is set to a large value to closely match the theoretical model. We show the results in Figure 3.10 and observe that the simulator results precisely track the theory including priority inversion points and delay values barring QoS_l's delay, which is slightly higher in the simulation. We believe that this is a result of the packet nature of the simulator versus the fluid model used in theory.

Two experiment setups are common in our simulator-evaluation: (1) a 3-node setup where two clients send RPCs to the same destination server for microbenchmarks, and (2) a 33-node or 144-node setup with an all-to-all traffic pattern where each host sends RPCs to the other hosts with an average and burst load of 0.8 and 1.4 respectively (similar to Figure 3.7) with Poisson arrivals. Setups that differ from above are described for corresponding experiments. QoS weights for experiments with 2-QoS levels are set at 4:1 and with 3-QoS levels are set at 8:4:1.

3.6.2 SLO Compliance

To evaluate how closely Aequitas' distributed admission control tracks per-QoS RNL SLOs, we start with the 3-node topology where two RPC channels, each running on a different host, issue 32KB WRITE RPCs on QoS_h to a destination server. To make the network persistently overloaded, each host issues RPCs at line rate with 70% of its RPCs at QoS_h and 30% of its RPCs at QoS_l.

[6] https://github.com/SymbioticLab/Aequitas

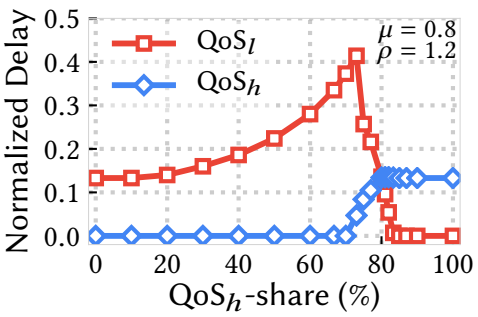


Figure 3.10: Simulated WFQ delay bounds with QoS_h:QoS_l weights = 4:1.

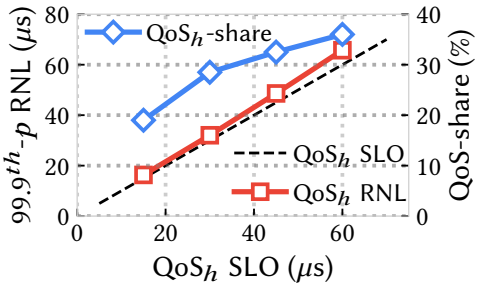


Figure 3.11: Aequitas provides SLO-compliance: achieved RNL closely tracks SLOs.

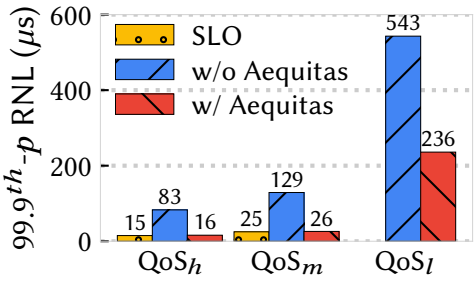


Figure 3.12: Aequitas significantly improves RNL, closely tracking the SLOs.

Figure 3.11 shows that Aequitas tracks the SLO target in terms of the 99.9th-p RNL for QoS_h extremely well as it is varied from 15μs to 60μs. The tradeoff between SLO targets and admitted traffic is also evident, with stricter SLOs resulting in fewer RPCs admitted on QoS_h.

We move onto the 33-node setup to illustrate that Aequitas preserves its ability to closely track RPC latency SLOs under different communicate patterns at a larger scale. We set the input *QoS-mix* of (QoS_h, QoS_m, QoS_l) to be (0.6, 0.3, 0.1) and show the achieved RNL at the 99.9th-p w/ and w/o Aequitas with the SLOs selected as 15μs and 25μs for QoS_h and QoS_m, respectively, in Figure 3.12. We treat QoS_l as the scavenger class as discussed in §3.5.

An interesting, and perhaps surprising, observation is that with Aequitas, the RNL of QoS_l reduces as well, i.e., **Aequitas is not a zero-sum game for per-QoS latencies.** The reasoning behind this is similar to the result in Reference [36] where all jobs can improve their performance with right prioritization. Given the improved RNL for QoS_h and QoS_m, RPCs in QoS_l have fewer RPCs to contend with, which as per Little's Law [138] implies that they will finish quicker as well. To verify, we collect the number of outstanding RPCs and show in Figure 3.13 that the decrease in instantanous outstanding RPCs in QoS_h + QoS_l indeed outweighs the increase in QoS_l outstanding RPCs, especially at the tail.

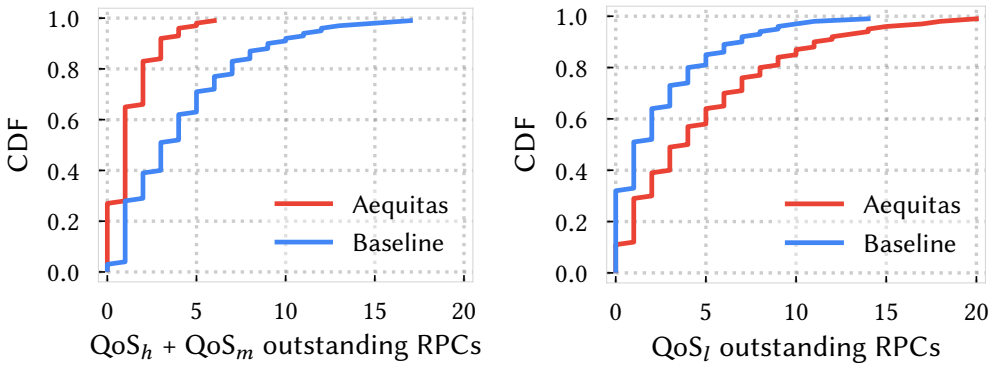


Figure 3.13: Comparison of number of outstanding RPCs per switch-port before and after Aequitas.

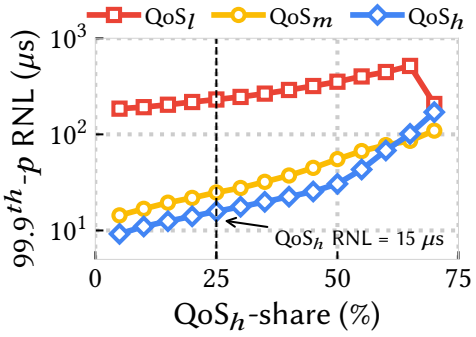


Figure 3.14: Baseline (w/o Aequitas) 99.9th-p RNL observed as QoS_h-share is varied.

3.6.3 Maximizing Admitted Traffic within SLOs

While SLO-compliance is the main correctness criterion, it can be met *trivially* by admitting a tiny amount of traffic. We show that Aequitas admits close to maximal traffic while retaining SLO-compliance, irrespective of the input *QoS-mix*.

To figure out the maximal admissible traffic associated with a given SLO, we measure 99.9th-p RNL in the 33-node setup without Aequitas as we vary QoS_h-share from 5 to 70%, keeping QoS_m at 25% and allotting the rest to QoS_l, as shown in Figure 3.14. We set the SLO for QoS_h at 15μs which corresponds to QoS_h-share of 25%, and SLO for QoS_m at 25us. Thus, in this setup, 25% is the maximal amount of traffic we can admit as admitting any more traffic will violate the correctness criterion of SLO-compliance. We then vary the input *QoS-mix* and plot both RNL and admitted *QoS-mix* in Figure 3.15. We can see that Aequitas converges closely to the maximal QoS_h-share while retaining SLO-compliance.

Further, we observe that Aequitas' algorithm is *self-consistent*, i.e., if the input *QoS-mix* is same as target, then very little traffic gets downgraded. The corollary of this result is that Aequitas helps solve the *race to the top* problem defined in §3.2.3 by effectively controlling the *QoS-mix*

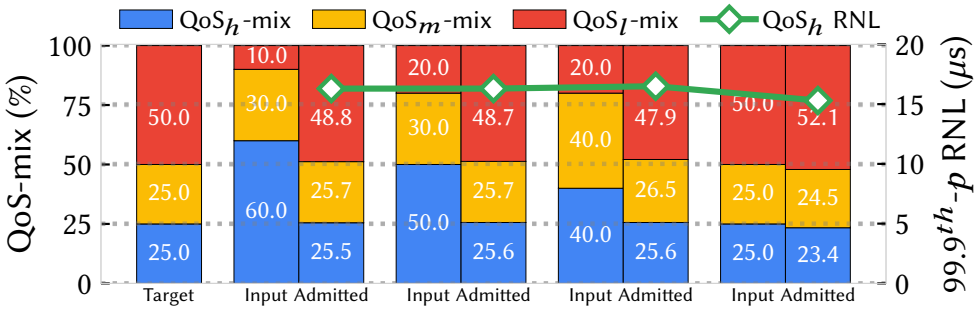


Figure 3.15: Aequitas admits close to maximal traffic while retaining SLO-compliance irrespective of input QoS-mix.

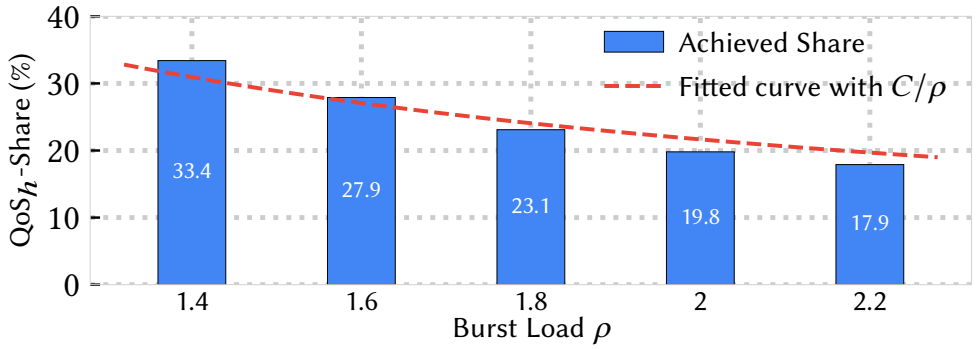


Figure 3.16: Aequitas adjusts admitted traffic that is inversely proportional to traffic burstiness.

independent of the input distribution.

3.6.4 Impact of Burstiness on Admitted Traffic

As we discussed in in §3.5.2, for a given set of SLOs, as burstiness of the traffic increases, the amount of traffic for which those SLOs can be provided for decreases. In Figure 3.16, we vary the burst load, ρ, and plot the QoS_h-share that Aequitas admits. While the simulation differs from the theoretical model in many ways such as packet-level behavior and congestion-control, we can observe, via the fitted curve in the plot, the inverse proportionality of admitted traffic w.r.t. burst load as per the theoretical formulation.

3.6.5 Fairness

Besides providing SLO guarantees, fairness is also a key goal in Aequitas' admission control. To evaluate if Aequitas ensures fairness across RPC channels, we modify the 3-node experiment in §3.6.2 such that Channel A issues 40% of its RPCs on QoS_h (equivalent to 40 Gbps worth of RPCs

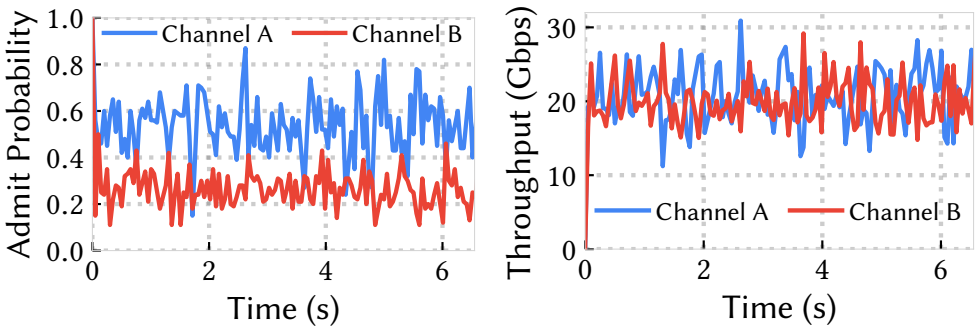


Figure 3.17: Admit probability and throughput of two RPC channels sending 80Gbps and 40Gbps QoS_h traffic with QoS_h SLO set to 15μs.

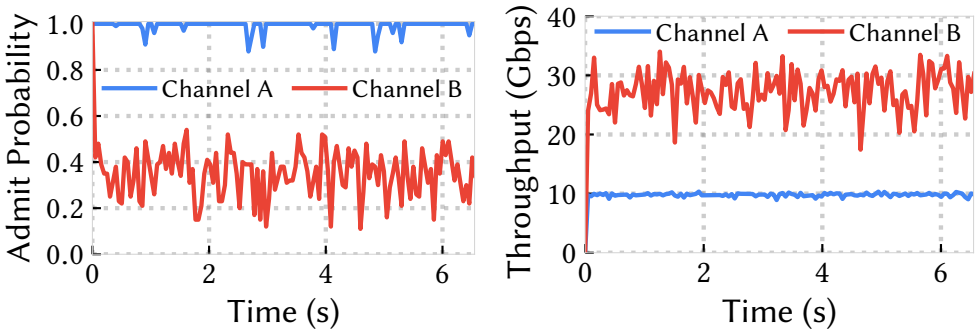


Figure 3.18: Aequitas maintains a near 1.0 admit probability for in-quota RPC channels that have demand less than fair-share. Excess quota is reclaimed by other channels to provide max-min fairness.

at 100Gbps link as the per-channel load is 1.0) whereas Channel B issues 80% of its RPCs on QoS_h. We set QoS_h SLO to be 15μs such that fairness implies that each channel gets 20% of its RPCs admitted to QoS_h and the rest downgraded. Figure 3.17 shows that Aequitas achieves fairness by converging to different values of admit probability for each individual channels.

Another important aspect regarding fairness is how Aequitas behaves when a channel is operating within its quota, i.e., its demand for QoS_h is below its fair-share. The expectation is that such a well-behaved RPC channel experiences minimal to no downgrades while continuing to meet the SLOs. We modify the above experiment in that Channel A issues only 10% of its RPCs on QoS_h (lower than its fair share of 20%) and plot the admit probabilities and achieved throughput in Figure 3.18. We find that Aequitas not only maintains admit probability of Channel A close to 1.0, it also allows Channel B to reclaim the excess, providing max-min fairness. We observe Channel A is able to consistently achieve a throughput of 10Gbps with 1^{st}-p p_{admit} = 0.82.

α and β are key parameters in that they posit a tradeoff between SLO-compliance and stability.

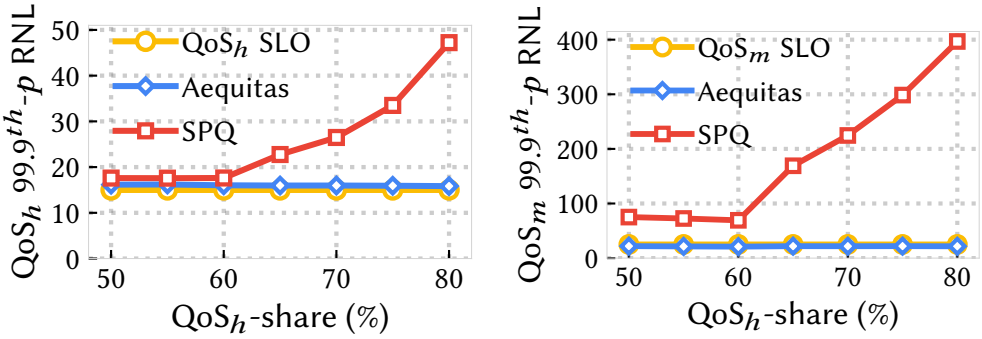


Figure 3.19: Aequitas compares with Strict Priority Queuing (SPQ) in providing SLO guarantees.

Results of a sensitivity analysis on α and β can be found in Appendix B.3.

3.6.6 Convergence Time

Convergence time affects how quickly Aequitas' algorithm achieves max-min fair-sharing amongst channels while ensuring SLO-compliance. Convergence time for both channels is 10ms in Figure 3.17 and 3ms in Figure 3.18. As above, α and β are key parameters here and we choose them to favor SLO-compliance.

3.6.7 Comparison with Strict Priority Queuing

Although strict priority queuing (SPQ) is not widely deployed in many production networks, it is widely used in literature to implement optimal scheduling such as SRPT [25, 160]. We evaluate how using SPQ alone handles network overloads by applying the same setting in our 33-node setup and replacing the underlying WFQs with SPQs. We fix the QoS_m distribution at 20% and increase the percentage of QoS_h traffic as shown in Figure 3.19. We observe SPQ fails to maintain predictability as more applications mark their RPCs as QoS_h; meaning, it does not resolve the *race to the top* problem.

3.6.8 Handling Different RPC Sizes

We now evaluate how Aequitas handles different RPC sizes as discussed in §3.5.1. We conduct an experiment where half the channels continue to issue 32KB RPCs while the other half issue 64KB RPCs. Figure 3.20 shows that Aequitas continues to meet the *normalized* RNL SLOs with different RPC sizes.

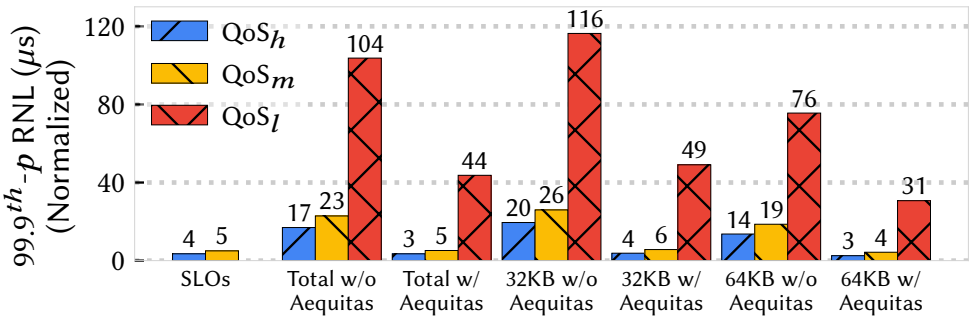


Figure 3.20: Aequitas uses RPC size to normalize latency with a non-uniform size distribution in a 33-node cluster.

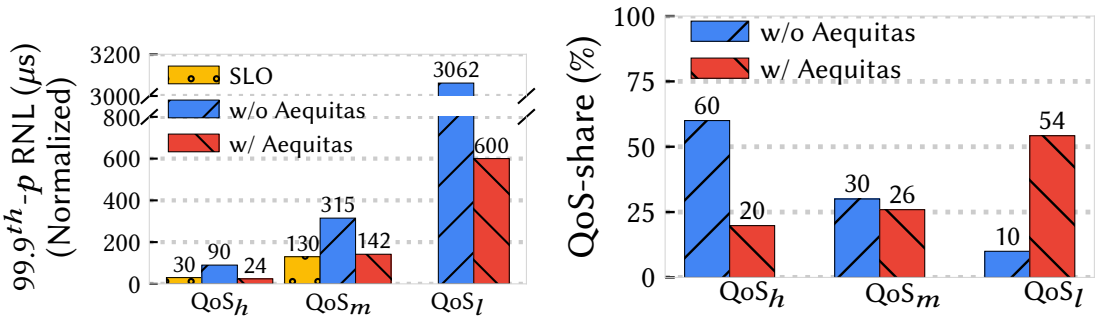


Figure 3.21: Aequitas' performance in a large (144-node) topology with production RPC sizes.

3.6.9 Large-Scale Eval with Production RPC Sizes

We evaluate Aequitas at a larger scale in a simulated 144-node topology, with RPC sizes taken from production, under extreme overload where we increase the burst load such that the maximum instantaneous load on the link is 25× its capacity. Figure 3.21 shows that Aequitas improves tail RNL in QoS_h/QoS_m by 3.7×/2.2× in this workload, continuing to meet RNL SLOs even under extreme overloads that can occur in production. We observe 20ms convergence time before the 99.9th-p latency becomes stable.

3.6.10 Comparison with Related Works

We compare Aequitas with four related systems, pFabric [25], QJump [89], D^3 [219], PDQ [96], and Homa [160], each with a complete implementation in our packet-level simulator. We use our production RPC size distribution with 50%/30%/20% input QoS-mix in the 33-node setup. We start with normalized SLO targets; for D^3 and PDQ, which do not consider RPC sizes in their design, these translate to 250us and 300us deadlines for QoS_h and QoS_m RPCs based on the average of production RPC-size distribution, respectively.

We record per-QoS 99.9^{th}-p RNL and percentage of traffic that meet the SLO targets from their initially assigned QoS levels. We also record network utilization, which we define as the achieved goodput divided by the maximum goodput based on the input arriving rate. Figure 3.22 summarizes the results and we make the following observations. First, Aequitas achieves the highest amount of traffic meeting SLO targets (QoS_m results, not shown, are similar). Second, Aequitas achieves better 99.9^{th}-p RNL for QoS_h and QoS_m than pFabric and QJump, which are SLO-unaware. D^3 and PDQ observe good performance on RNL, however, a lower percentage of traffic meets the SLO targets / deadlines. This is because both D^3 and PDQ terminate an RPC early when it cannot meet its deadline, which also lowers network utilization down to nearly 50%. pFabric favors short RPCs using SRPT scheduling, but doesn't meet SLO targets for large RPCs, which can be equally important. Homa also adopts SRPT, but its usage of dynamic in-network priorities favors even more applications' small RPCs, leaving more large RPCs' SLO goals to be ignored. QJump provides good performance at the packet level by rate-limiting higher QoS traffic at end-hosts, however at RPC level, Aequitas' performance is better both in terms of RNL and percentage of traffic meeting SLOs.

3.6.11 Testbed Evaluation

Our prototype implementation of Aequitas is built in a production RPC stack, and it incorporates both Phase 1 and Phase 2 of Aequitas' design. Aequitas' algorithm computes an admit probability per RPC channel, which is mapped to multiple per-QoS TCP sockets. On RPC completion, the RNL measurement is fed into the Aequitas' algorithm.

We deployed the prototype in a 20-machine testbed with 100Gbps NICs connected to a single switch that supports ~10 QoS queues with configurable weights. We set the QoS_h:QoS_m:QoS_l weights to 8:4:1. Each machine issues 32KB WRITE RPCs to other machines in an all-to-all communication pattern. To circumvent the issues in RNL measurements described in §3.2.2.1, we provision enough CPU such that the elevated network-latency measurement is purely a result of network overload.

Figure 3.23 shows the RNL SLOs and QoS-mix w/ and w/o Aequitas. The setup is similar to §3.6.3 with 3 QoS levels and an input QoS-mix at (0.5, 0.35, 0.15). The SLOs are set as per a QoS-mix of (0.2, 0.3, 0.5). For confidentiality reasons, we show normalized[7] RNL measurements and find that Aequitas meets its promise of achieving SLO targets.

[7]We normalize each QoS level with their observed 99.9^{th}-p RNL when input QoS-mix is same as the target QoS-mix (0.2, 0.3, 0.5).

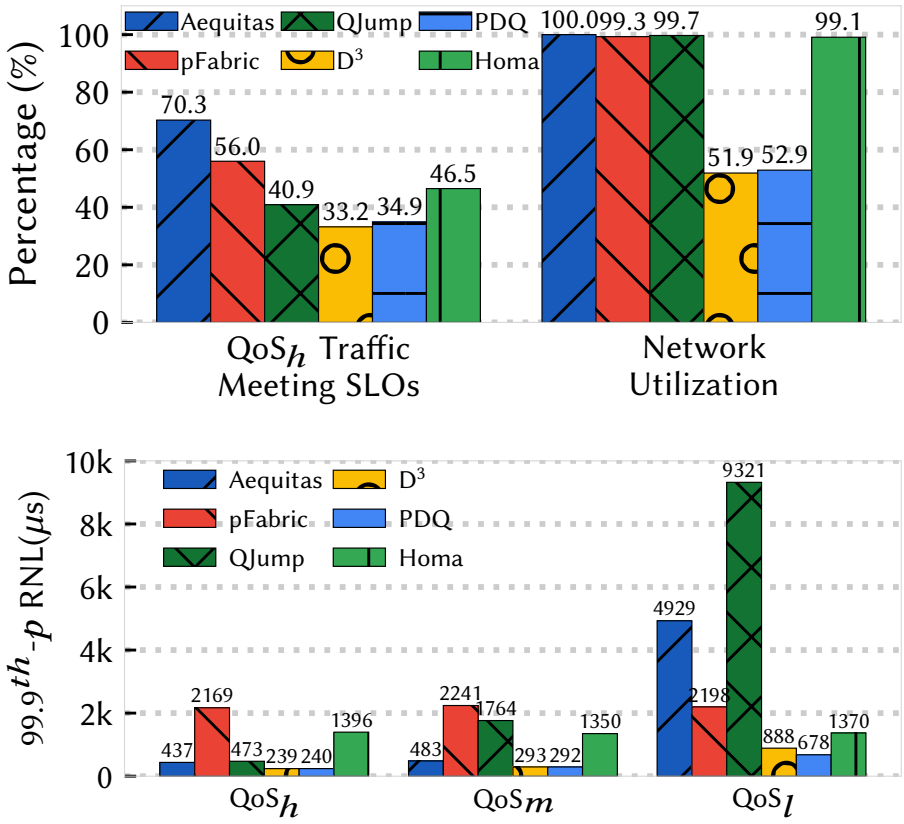


Figure 3.22: Aequitas compared with related works in the simulated 33-node setup with production RPC size distribution.

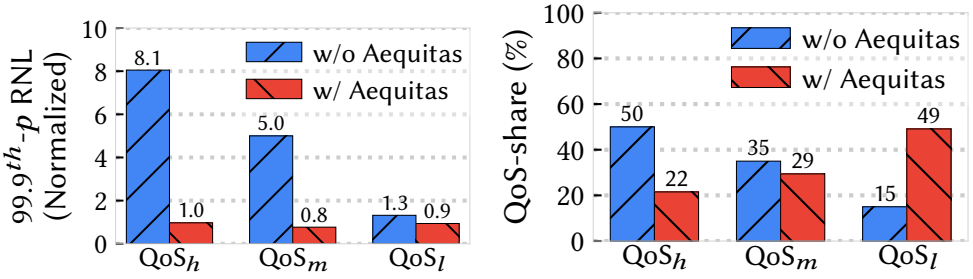


Figure 3.23: Result from testbed implementation shows that Aequitas maintains SLO compliance and converges to the target QoS-mix.

3.6.12 Results from Production Deployment

We now present results from Aequitas' production deployment with a focus on Phase 1 deployment. Phase 2 results are not ready to be collected as the chapter is written. Figure 3.24 shows the results (collected from a random sampling of 50 clusters) from our fleet-wide deployment of Phase 1, with

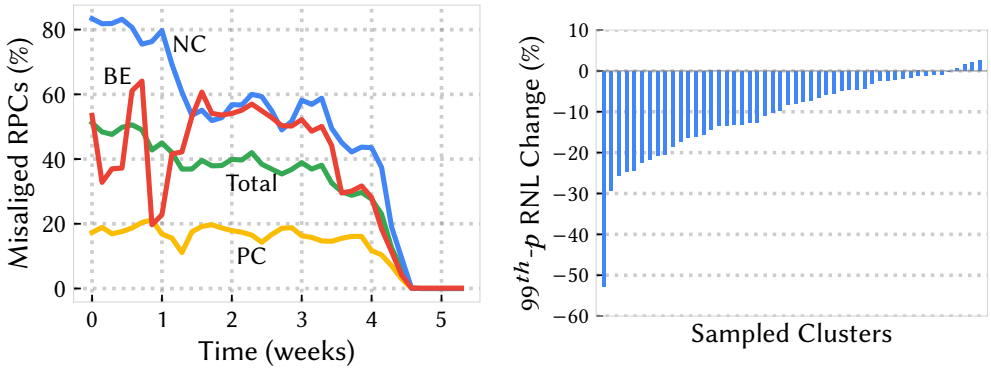


Figure 3.24: Production deployment of Phase 1 of Aequitas shows improvement in QoS-misalignment and 99th-p RNL.

two important metrics:

1) Misalignment percentage between RPC-priority and QoS, i.e., percentage of *PC* RPCs flowing in QoS_l and QoS_m queues, *NC* RPCs flowing in QoS_h and QoS_l queues, and *BE* RPCs flowing in QoS_h and QoS_m queues. Aequitas brings misalignment down from up to 80% to nearly zero.

2) Aequitas achieves up to 53% reduction in 99^{th}-p RNL for high priority traffic. RNL was measured as described in §3.2.2.1 for Storage READ and WRITE RPCs. A small number of clusters showed a minor regression primarily because of initial highly skewed QoS distribution, competing traffic that is not yet aligned through Aequitas, and changes in traffic pattern during the measurement window.

3.7 Related Work

Packet Scheduling: Packet scheduling is a classic networking topic that focuses on different variations of weighted fair queuing (WFQ) in switches, routers, and middleboxes [65, 176, 38, 85, 88, 199, 217, 162]. Aequitas builds upon this pioneering work and extends the analysis by exploring the interactions among QoS levels as well as WFQ's admissible regions to provide SLOs for higher-priority RPCs. Centralized packet arbitration solutions [177, 171] could provide latency guarantees, but are difficult to deploy at scale in datacenters.

Flow and Coflow Scheduling: Flow scheduling algorithms in datacenters focus on minimizing the average or tail flow completion time (FCT), typically by prioritizing short flows [25, 33, 80, 160, 76]. Aequitas works at the RPC granularity, provides guaranteed SLOs for RPC network-latency, and uses WFQ mechanism versus strict priority. Coflow scheduling minimizes the average coflow completion time (CCT) instead of FCT [53, 52, 67, 21], but it does not capture RPC semantics either.

Network Calculus-Based Scheduling: Network calculus has been widely used as a tool to provide worst-case latency guarantees by many related works [208, 59, 191, 104, 235, 233, 234]. However, none highlighted the interactions between QoS levels and the possibility of priority inversion. PriorityMeister [235], SNI-Meister [233], and WorkloadCompactor [234] target tail latency SLOs, but they all base their analysis on SPQ and rely on prior knowledge of stationary traffic traces.

AQM and QoS Solutions: Active Queue Management (AQM) performs admission control by probabilistic dropping at the packet layer to prevent congestion under over-subscription while approximating fairness [78, 175, 174, 225, 167]. Aequitas uses a similar probabilistic admission control, but at the layer of RPCs. QoS-based solutions such as IntServ [220] and DiffServ [41] provide applications better service relative to *BE* traffic running on the Internet. Aequitas is similar to DiffServ in that it also differentiates traffic at the edges as per their priority, but focuses on datacenter environments to provide guaranteed latency SLOs for RPCs.

Congestion Control: Datacenter congestion control solutions have focused on quickly achieving max-min fairness with or without hardware support and using edge-based (delay-based) and network-supported (e.g., using ECN or programmable switch) mechanisms [24, 238, 135, 156, 125, 160]. Aequitas, which operates at the RPC layer, relies on a well-functioning congestion control algorithm at the transport layer to keep switch buffer occupancy small, alleviate packet-losses, and fully utilize available bandwidth.

Network Bandwidth Sharing: Over the last decade, another prominent direction of research has been network bandwidth sharing in public and/or private clouds [51, 91, 106, 197, 179, 34, 35, 180, 162]. Similar solutions at the WAN level include, among others, BwE [124] and SWAN [97]. Aequitas differs from these efforts by focusing on providing isolation in terms of RPC latency to critical RPCs.

Server-Side RPC Overload Management: Another line of work is to provide overload management at the layer of RPCs or request/responses but focuses on server-side overload such as CPU contention [50, 218, 232, 79]. Aequitas complements such schemes by focusing on network overload, providing guarantees on RPC network-latency.

3.8 Conclusion

Today, developers running in shared multi-tenant cloud environments have no effective way to provide RPC latency SLOs. In this chapter, we take an important step toward this higher-level goal by providing a first solution for SLOs for the network component of RPC latency. We present the design, implementation, and evaluation of Aequitas, built around the observation that dynamically mapping RPCs to widely-available network QoS classes can bound RPC network-latency in a shared environment with no centralized control. We employ Network Calculus-based analysis to set the

ratio of RPCs admitted into the network at different priority levels with the goal of guaranteeing quantitative tail latency targets for all priority classes except for the lowest (best-effort) class. We hope our work will inspire other techniques to comprehensively deliver RPC latency SLOs as a fundamental building and reasoning block for distributed systems developers. On the theoretical side, we leave an open question: what are the closed-form delay equations for an arbitrary number of QoS levels, if any? Solving this challenge, or proving non-existence of such generalization, will provide valuable insights in designing future QoS-aware systems.

CHAPTER 4

Vulcan: Automatic Query Planning for Live ML Analytics

Both Justiita and Aequitas explored QoS inside the datacenter. We now further extend our QoS research interests from within the datacenter into applications that span both datacenter and edge devices. This chapter introduces Vulcan, a system we build to provide better QoS when serving live ML analytics. It turns out that serving live ML analytics is not an easy task, and it involves many steps with room for optimization. This leads to another challenge we face in this work – optimization cost in the course of finding the optimal solution, besides the goal of better QoS with high utilization.

The remaining of this chapter is organized as follows. Chapter 4.1 introduces Vulcan. Chapter 4.2 describes the existing workflow of how live ML queries are served, and why they need better QoS support. Chapter 4.3 and Chapter 4.4 describe the system overview and design details of Vulcan. How Vulcan performs online adaptation for deployed queries are described in Chapter 4.5. We then discuss our evaluation results in Chapter 4.6, followed by a discussion of existing research in Chapter 4.7 and a conclusion on Vulcan in Chapter 4.8.

4.1 Introduction

Recent years have witnessed a growing demand for machine learning (ML) analytics. *Live ML analytics* – with applications in edge-assisted autonomous driving [196, 227, 228], live traffic analysis [6, 143], and real-time speech recognition [10, 55] – stands out due to its large-scale deployments [6, 8, 7]. Live ML analytics involves ML pipelines at its core, where each pipeline consists of a series of operators to perform specific ML tasks. For example, an autonomous driving perception query that detects surrounding objects of an autonomous vehicle may contain filtering operators for road surface removal [86] and 3D data compression [169, 82], along with a 3D object detector to perform object detection [130, 223].

Live ML analytics differentiates itself from ML on stored data in two key characteristics. First, live ML pipelines are deployed across *heterogeneous infrastructure*, spanning multiple tiers such as device edges, on-premise edges, public MEC, and cloud datacenters [6, 8, 9, 12]. Second, live ML queries have *latency requirements* besides accuracy targets as the analytics is based on real-time data. Some analytics, such as object detection in autonomous vehicles, may have more stringent requirements than others depending on the priority of the task. Therefore, before deploying live ML analytics, each query describing the ML task must go through careful *query planning*. Specifically, this involves (i) constructing the pipeline by selecting a series of operators and ordering them; (ii) determining the physical placement of pipeline operators across infrastructure tiers; and (iii) selecting configurations of the pipeline operators, to optimize for query performance. A joint optimization of the three aspects is required for optimal performance and resource consumption.

Recent research on these topics have been piecemeal, focused on compute alone, and hence largely sub-optimal for live ML analytics. Although declarative query languages (e.g., SQL) have been proposed for ML queries, there exists no systematic approach to automatically construct pipelines based on query's end-to-end latency requirement. This includes selecting and ordering the filtering modules during pipeline construction, which have a great impact on the performance characteristics of ML pipelines. Furthermore, when choosing physical placement of pipeline components, one cannot afford to exhaustively search for the optimal placement through real-world deployments. As a result, deployments often rely on past experience with simple heuristics [221, 9]. While recent solutions have focused largely on selecting query configurations [226, 107, 39, 198, 103], they assume the ML analytics component to be a monolithic module instead of a pipeline. As such, they are profiled for compute alone, assuming they are running in a homogeneous datacenter. In the process, networking resources are ignored, and the complexities of multi-resource planning of compute and network are overlooked altogether. Finally, these solutions rely heavily on domain-specific insights of video content, which are not applicable to general ML scenarios beyond video analytics [226, 107, 145].

After deploying the query in the wild, one must also adapt the query plan based on runtime dynamics such as data content and/or resource changes [168, 39, 107, 172]. Prior solutions in providing online adaptation [107, 226] focus on data content changes alone but do not adapt to compute and network resource changes, which are common in edge environments [168]. An ideal solution should change query configuration *and* placement during online adaption.

In this work, we consider how to perform automatic query planning – i.e., constructing, placing, and configuring ML pipelines, along with adapting to runtime dynamics – for live ML queries based on user-provided performance requirements. The goal is to find query plans that optimize latency and accuracy, while minimizing the network and compute resource consumption. Generating such query plans, however, is challenging as jointly optimizing pipeline construction, placement,

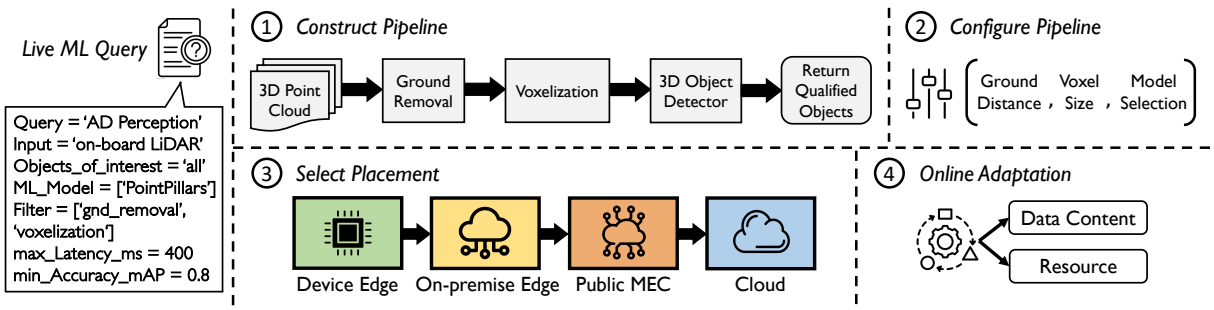


Figure 4.1: The existing workflow of query planning for a live ML query.

and query configuration leads to a much larger search space. Moreover, as query performance is resource and data dependent, considering all possible resource and data dynamics that may or may not happen in the future can explode the search space. Clearly, a more efficient online adaptation technique is needed for faster convergence and lower cost.

We present Vulcan, an ML analytics system that performs automatic query planning for live ML queries. Its design includes the following key ideas to overcome the aforementioned challenges:

(1) Vulcan defines a novel metric to quantify each filtering operator in the pipeline by combining query precision, recall, resource usage, and latency (§4.4.2). This converts the complexity of filter ordering from exponential to linear.

(2) Vulcan carefully identifies components of ML pipelines that are *independent* of placement. This allows Vulcan to dramatically prune the search space of placement options.

(3) Vulcan efficiently explores the best combination of configuration knobs using Bayesian Optimization (BO) [158, 43]. It designs BO's priori assumptions and acquisition function to jointly optimize placement and configuration selection.

(4) Vulcan adapts quickly to dynamic changes in data and resources by (i) designing programming interfaces that allow for dynamic updates to live pipelines modules without disrupting them, and (ii) leveraging prior knowledge to make faster decisions on modifying configurations and placement.

We evaluate Vulcan using real-world datasets on a wide range of applications including traffic monitoring, autonomous driving perception, and automatic speech recognition. Experiments are conducted under an edge hierarchy that represents real compute and network resource setting of our production infrastructure. Vulcan generates query plans with better profiling cost by 4.1×-30.1× over state-of-the-art ML analytics systems while delivering 3.3× better query latency performance. Vulcan outperforms existing solutions for ML query configuration and placement selection with up to 2.8× better query latency and 174× lower network resource consumption. Vulcan also achieves consistently better 99^{th}-p latency by up to 2.5× by adapting to both data and resource changes during online adaptation (§4.6).

In summary, we make the following research contributions:

- We provide an end-to-end system design for live ML queries for a wide range of real-world ML applications.
- We propose novel solutions on search space reduction for constructing, placing, and configuring ML pipelines.
- We implement interfaces and control loop for online adaption for fast re-profiling and dynamic re-configuration.

4.2 Background and Motivation

We start with an overview of the workflow of live ML query processing, followed by motivating examples to highlight the key aspects when choosing query plans.

4.2.1 Processing Live ML Queries

We explain the query planning workflow by walking through an example query which detects surrounding objects in edge-assisted autonomous driving, as shown in Figure 4.1. The input query specifies input data, object of interests, pipeline operators, and performance requirements on accuracy and latency.

① **Constructing the pipeline.** The first step is constructing the ML pipeline by choosing a series of operators to perform the task. Specifically, it involves choosing *filtering operators*, such as voxelization and ground removal, to be deployed for substantial resource efficiency prior to the ML models, such as the 3D object detector [130, 223] in the case of Figure 4.1. The ordering of the filtering operators has a significant impact on performance. The best ordering is dependent on the data, resource availabilities, and performance requirements.

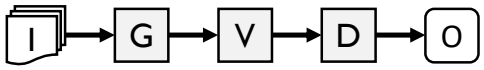
② **Configuring the pipeline.** After the operators are chosen in the pipeline, the next step is selecting *configuration knobs* to achieve the best tradeoff between query performance and resource consumption [226, 107, 98, 39, 145]. In our example, configuration knobs include ground distance, the voxel size, and the choice of a 3D object detection model.

③ **Selecting physical placement.** The next step is placing pipeline operators across the heterogeneous edge infrastructure, starting from the device edge and to the cloud (Figure 4.1). This heterogeneity introduces complexity in placement of the operators and influences the *end-to-end latency performance* of the ML pipeline.

④ **Performing online adaptation.** After a query is deployed, its performance is affected by *runtime dynamics* due to resource changes and data variations. Resource changes are common

A_{req}	L_{req} (ms)
0.8	400

(a) Performance Requirement



(b) Pipeline

Config	A	L (ms)	Size (MB)
C_1	0.85	260	6|4.4|3.2
C_2	0.8	270	6|3.6|2.2

(c) Query Configuration Results

Figure 4.2: (a) Performance requirement on accuracy (A_{req}) and latency (L_{req}) of the example query. (b) Query's pipeline ($Input \rightarrow GroundRemoval \rightarrow Voxelization \rightarrow ObjectDetector \rightarrow Output$). (c) Offline profiling results. The last column records the size of the data at different stages of the pipeline: data at the source | after 'G' | after 'V'. Data after 'D' is not shown due to negligible size.

because of other workloads on the edge infrastructure (e.g., 5G RAN containers) or network outages [168, 136]. The content of the data, such as lighting or object densities [39, 107], can also change during the lifetime of a query. An ideal solution should adapt to runtime dynamics by adjusting the pipeline's filters, its configurations, and placement.

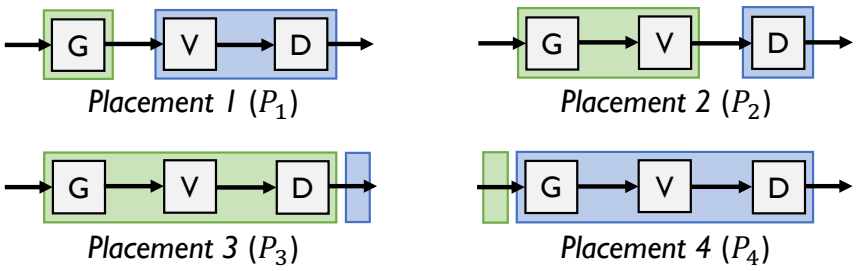
4.2.2 Motivating Examples

We now highlight some of the key challenges toward performing query planning for live ML queries using toy examples.

Jointly optimizing configuration and placement. Current practice considers pipeline placement and query configuration separately [107, 39, 115, 221, 153]. Designers first perform offline profiling to determine the optimal query configuration, and then select placement based on heuristics (prioritize network or compute, etc.) or greedy algorithms [221]. Such an approach fails to consider additional query latency introduced by pipeline placement, leading to sub-optimal query plans.

To illustrate this, take an example autonomous driving query, whose performance requirements and pipeline are shown in Figure 4.2a and 4.2b. Figure 4.2c records the offline profiling results assuming only 2 sets of configuration knobs (C_1 and C_2) exist. In this case, the baseline approach identifies C_1 as the optimal configuration since it performs better both in terms of accuracy and latency than C_2. It then pairs C_1 with one of the placement choices in our simplified two-tier infrastructure shown in Figure 4.3a.

However, as shown in Figure 4.3b, none of placement choices would satisfy the query's end-



(a) Placement Choices

Config	Placement	*A*	*L* (ms)
C_1	P_1	0.85	480
C_1	P_2	0.85	420
C_1	P_3	0.85	520
C_1	P_4	0.85	560

Config	Placement	*A*	*L* (ms)
C_2	P_1	0.8	450
★ C_2	P_2	0.8	380
C_2	P_3	0.8	540
C_2	P_4	0.8	570

(b) End-to-end Performance Results

Figure 4.3: Placement choices and corresponding end-to-end performance. (a) All four feasible placement choices in a two-tier setting (Device Edge→Datacenter). (b) The baseline approach which first acquires the optimal combinations of configuration knobs during offline profiling (C_1) and then selects placement fails to meet the latency target. An ideal solution should select C_2 & P_2 by jointly optimizing configuration and placement. Note accuracy does not depend on placement and remains unchanged.

to-end latency requirement if C_1 is selected. The end-to-end latency is composed of the compute latency (time spent in the object detector) and additional network latency occurred over the edge. For example, the additional network latency in P_1 is computed as the time it takes to transmit the output data of the ground removal module using the link bandwidth, which is set to 20MBps in this example.[1] An ideal solution should select C_2 by jointly considering both placement and configuration.

Constructing pipelines based on performance requirements. Different orderings of filtering operators lead to different performance characteristics of ML pipelines. Therefore, an ideal solution must construct the pipeline based on query-specific performance requirements. For instance, if we change the performance requirements of the original example query to a lower latency target with more tolerance on accuracy (shown in Figure 4.4a), then none of the query plans in Figure 4.3b can satisfy the new latency target as long as they use the pipeline from Figure 4.2b. Instead, we need to use a new pipeline which swaps the order of the two filters (Figure 4.4b). As filters are not

[1]In this example, the compute latency is assumed to be doubled when placing the detector on the device edge (i.e., in P_3).

A_{req}	L_{req} (ms)
0.6	300

(a) Performance Requirement

(b) Pipeline

Config	A	L (ms)	Size (MB)
C_1	0.7	180	6|4|2.7
C_2	0.6	190	6|2.8|2

(c) Query Configuration Results

Config	Placement	A	L (ms)
C_2	P_2	0.6	290

(d) Optimal Query Plan

Figure 4.4: Given an updated performance requirement, a new pipeline is required to meet the latency target. The new pipeline swaps the order of the two filters ('V' & 'G'), delivering a different performance characteristic than the old one in Figure 4.2b.

independent to each other, placing 'V' before 'G' reduces more data, leading to better latency but lower accuracy (Figure 4.4c). Figure 4.4d shows the end-to-end results with the optimal query plan using the new pipeline. We omit for brevity the same process of finding the optimal placement and configuration as we did earlier.

Building an ideal solution that handles all the aforementioned aspects of live ML query planning is non-trivial, as selecting the right pipeline, placement, and configuration jointly leads to a huge search space both during offline profiling and online adaptation. We next describe how Vulcan overcomes these challenges in a high-level system overview.

4.3 System Overview

Vulcan is an ML analytics system that provides automatic query planning for live ML queries. It takes charge of the entire lifecycle of a ML query by constructing, configuring, and placing its ML pipeline, and performing online adaptation after the query is deployed.

Figure 4.5 presents a high-level system diagram of Vulcan. A user launches a live ML analytics task by submitting an input query to Vulcan, along with performance requirements. An example of Vulcan input queries can be found in Figure 4.1. Upon parsing the query, Vulcan Profiler generates its query plan by determining the query pipeline, placement of pipeline operators, and pipeline configuration. In Vulcan, query plans are evaluated using a utility function we define that combines the query performance and resource consumption (§4.4.1). To determine which pipeline to use for a query, Vulcan first constructs an initial pipeline by mapping user query specification to a general template optimized for performance and resource efficiency, and then determines the best ordering

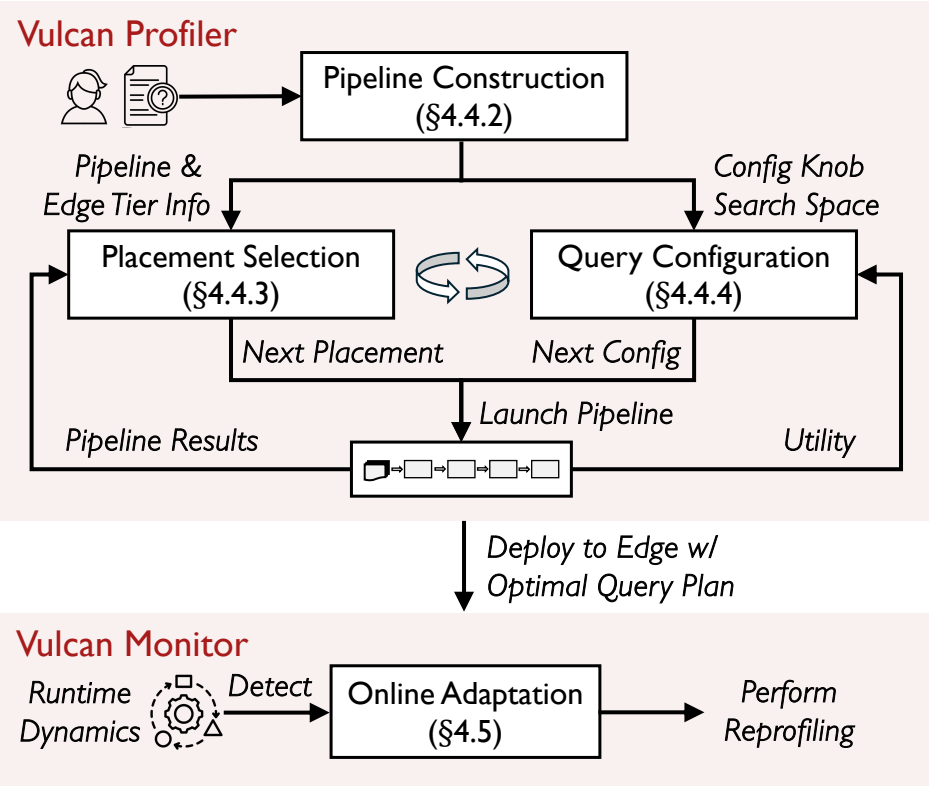


Figure 4.5: High-level workflow of Vulcan.

of filtering operators based on a new metric we define to capture the impact of filters on query latency and accuracy (§4.4.2).

Given a constructed pipeline, Vulcan jointly searches for the best placement and query configuration which, when combined, gives the highest utility. To explore placement choices with low cost, Vulcan reuses intermediate results from pipeline runs, such that a pipeline with the same configuration only needs to be offline profiled once. In the meantime, it also early prunes unpromising placement choices to further reduce the profiling cost (§4.4.3). For each placement, Vulcan searches for the best query configuration by leveraging Bayesian Optimization to explore a large number of query configurations with a small number of trials (§4.4.4).

After query deployment, Vulcan continues to monitor query performance to detect runtime dynamics. During such events, Vulcan reprofiles the pipeline in a quick and low-cost fashion by leveraging prior knowledge (§4.5).

4.4 Vulcan: Profiler Design

This section introduces the utility function we define to compare query plans, followed by how Vulcan construct, place, and configure the ML pipelines for live ML queries.

4.4.1 Defining Utility of Query Plans

We start by defining a utility function to evaluate the value of a query plan as we explore the search space. Existing literature has proposed various utility functions [27, 108, 154, 193] to combine query accuracy and latency. We build on top of these works and extend the utility function introduced in VideoStorm [226] to make it *resource-aware*. Resource consumption for live ML queries is as important as query performance, because each ML pipeline is deployed over edge infrastructure, where limited network and compute resources must be shared between applications. A query plan with good performance but excessive resources is undesirable.

Given a pipeline q with placement p and pipeline configurations c, we define $U_{q,p,c}$, the utility function of a query plan, as the ratio of the query performance to resource consumption:

$$U_{q,p,c} = P_{q,p,c}/R_{q,p,c} \tag{4.1}$$

such that the higher the utility value is, the better performance and cost for the query plan. $P_{q,p,c}$ combines query accuracy (A) and end-to-end latency (L) by calculating the reward (penalty) for achieving good (bad) performance based on a minimum accuracy target (A_m) and a maximum latency target (L_m):

$$P_{q,p,c}(A, L) = \gamma \cdot \alpha_A \cdot (A - A_m) + (1 - \gamma) \cdot \alpha_L \cdot (L_m - L) \tag{4.2}$$

, where $\gamma \in (0, 1)$. γ allows users to express their preference between accuracy and latency. $R_{q,p,c}$ combines the compute and network resource consumption of the pipeline:

$$R_{q,p,c} = \alpha_{gpu} \cdot R_{gpu} + \alpha_{net} \cdot R_{net} \tag{4.3}$$

The consumption of the compute (R_{gpu}) is calculated as the fraction of the GPU processing time used by the query. In Vulcan, we assume compute cost is dominated by GPU cost, as the queries we tackle rely heavily on GPU-based DNN models. The network resource consumption (R_{net}) is calculated as the sum of the fraction of the network bandwidth used by the pipeline on *each* network path between the edges. The constants α_A, α_L, α_{gpu}, and α_{net} are set by the operator to balance query performance and resource usage. Note that Vulcan's solution is orthogonal to the utility function and works for any utility function defined by the operator.

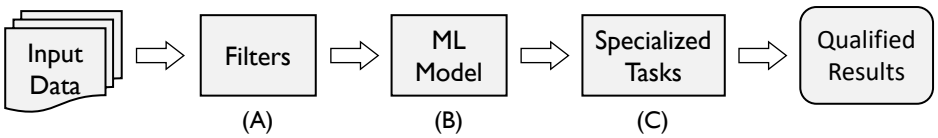


Figure 4.6: Template used by Vulcan to construct the initial pipeline.

4.4.2 Determining the Query Pipeline

The first step in profiling is to construct the query pipelines. Vulcan performs pipeline construction by first generating an initial pipeline, and then determines the best ordering of pipeline operators to carry to the later profiling stages.

4.4.2.1 Constructing the Initial Pipeline

Given a user query, Vulcan generates an initial pipeline using a general template with several types of building blocks, as shown in Figure 4.6. Starting from the data source, Vulcan constructs the initial pipeline by inserting (A) filtering modules which reduce the data size or data rate via sampling or filtering techniques, (B) the ML model to perform the actual inference task such as object detection, and (C) specialized modules for additional tasks required by the query, such as an object tracker (e.g., re-identification modules [110, 134, 137]), which can be performed only after the major ML inference task. Vulcan uses a pool of filter and ML modules that are readily available, provided by users, infrastructure providers, or third-party developers and organizations (e.g., public ML model zoos) to handle user queries. Filters and the ML model to use for the query is specified by the user in the input query (Figure 4.1). Based on the chosen operators, Vulcan generates a list of configuration knobs among which the profiler searches for an optimal set of configurations (§4.4.4).

The key insight behind arranging the building blocks in this way is to reduce the amount of data transfer across the edge *earlier* in the pipeline and leave operators with higher computation cost in *later* pipeline stages. This maximizes the savings in both network and compute resource as less data is transmitted and processed across the edge tiers. The design can improve end-to-end query latency by reducing the network latency as well as the GPU processing delay with potentially smaller data size for ML inference.

This initial pipeline leaves us with a follow-up given the impact of filter ordering on query performance (§4.2.2): *In what order should we place the filters?*

4.4.2.2 Selecting the Ordering of Filters

A naïve solution for selecting the filter ordering is to explore pipeline placement and configuration for all possible orderings; this, however, does not scale as the number of filters increases. In Vulcan, we propose a new solution that compares the impact of different filter orderings on query's accuracy, latency, and resource consumption by evaluating ***recall*** and ***precision*** of filters. We first explain how it works for a single filter before moving on to the multi-filter scenario. We define recall of a filter as the fraction of samples in the original data that contains the objects of interest (i.e., relevant data) that passes through the filter. On the other hand, precision of a filter is the fraction of data samples in its output that contains relevant data. For a given filter, we would expect its recall to be high such that it still captures most of the desired data, and query *accuracy* is preserved. A filter with low recall drops true positive samples which cannot be recovered later in the pipeline. Among filters with the same recall, we prefer the ones with higher precision because these filters provide higher data reduction rate by picking up fewer irrelevant samples, leading to better *latency and resource savings*.

We can now define the metric to evaluate how a given filter affects a query plan's utility ($U_{q,p,c}$). The metric should also handle a query's preference between latency and accuracy, based on the parameter γ defined in Eq (4.2) in our utility function. To this end, we leverage a variation of the F-measure in information retrieval theory [187] to encode this preference. Denote F_γ as the score for a given filter with its precision and recall measurements:

$$F_\gamma = (1+\beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}} \tag{4.4}$$

where $\beta = \gamma/(1-\gamma)$. In the F-measure definition, the value of β captures how many times recall is considered more important than precision. As we explained earlier, recall and precision corresponds to query accuracy and latency respectively, and thus the value of $\gamma/(1-\gamma)$ (see Eq (4.2)) is used to capture our accuracy-latency preference.

Measuring F_γ tells us how well a single filter fits into a query's optimal query plan. However, two challenges remain. First, directly applying F_γ to sort a sequence of filters does not work well as the recall of a filter can change based on its preceding filter. Second, filters may have their own configuration knob that leads to different precision or recall measurements. Applying one set of configuration for all filters oversimplifies the problem with inaccurate estimation, whereas evaluating too many configuration sets increases profiling cost.

We address multiple filters by treating a sequence of filters as a *bulk filter* with input being the data source and the output being the one from the last filter, and measure the overall F_γ using representative data for each permutation of the available filters. To deal with filters with various configurations, we choose a few representative configuration settings to capture the effect

of configuration knobs on filters, where in each setting all the filters are configured at the x-th percentile in the range of their configuration knob. Filters with no configuration parameter remain unchanged. We empirically find x = 20-th, 50-th and 80-th good enough at picking promising pipelines, which results in $\{3 \times$ total number of filter ordering$\}$ F_γ values to collect. Vulcan then picks the ordering which achieves the highest F_γ to complete the pipeline construction.

4.4.3 Determining Placement Choices

After constructing the pipeline, the next step is placing each of the pipeline operators across the edge infrastructure (see Figure 4.5). On the one hand, rule-based solutions are good at reducing search cost but may fail to explore all promising placement choices. On the other hand, exhaustively searching through all placement choices requires high search cost during profiling as we deploy the query across the edge. Vulcan combines the benefits of the two approaches by (i) reducing search cost by reusing intermediate results from pipeline runs, and (ii) early pruning unpromising placement choices.

4.4.3.1 Reusing Pipeline Results

The idea of reusing pipeline results is based on two key observations we make in live ML pipelines. First, query accuracy does *not* depend on the placement choices of a pipeline with the same query configuration. Second, the amount of data generated after each pipeline operator is independent of placement choices. These observations allow us to deploy the pipeline ***offline*** *in the datacenter* ***only once*** *per selected query configuration* during the profiling stage to collect pipeline operator results, and *reuse* those results to evaluate a new placement choice by calculating additional *latency and resource consumption components* introduced by the placement, while reusing the same query accuracy result. Given a total of M placement choices and N combinations of pipeline configurations, our solution improves the search complexity from $O(MN)$ to $O(N)$ for a given pipeline.

Algorithm 3 describes how Vulcan evaluates placement choices. Given a pipeline, we begin with generating a collection of all feasible placement choices. Obviously infeasible choices where operators are placed in a different order than they appear in the pipeline (e.g., placing the DNN model in front of the filters) are excluded from the collection. For each placement choice, Vulcan explores promising query configurations to evaluate the utility of the query plan (details of how Vulcan picks query configurations are in Section 4.4.4). For every new set of pipeline configurations, Vulcan launches the pipeline inside the datacenter. In this case, Vulcan not only collects query performance and resource consumption for utility calculation but also caches intermediate results from each pipeline operator, including the operator's output size, output bandwidth, and data

Algorithm 3: Vulcan Placement Selection

```
1  Notation:
     q: constructed pipeline (from §4.4.2.1),
     𝒫: all feasible placement choices given q,
     A_m: accuracy target, L_m: latency target, U: utility,
     res: pipeline results used to calculate utility
2  Function SelectBestPlacement(q, A_m, L_m):
3    𝒫 ← GenerateAllPlacementChoices(q)
4    foreach placement p ∈ 𝒫 do
5      c ← NextConfig()        ▷ from §4.4.4
6      count = 0               ▷ reset early pruning counter
7      if c.hasExplored() then
8        res = LoadFromCache(q, c)
9      else
10       res ← LaunchPipeline(q, c)
11       CachePipelineResults(res)
12     U_{p,c} = CalculateUtility(q, p, res)
13     if U_{p,c} ≤ U_{p,c}(A_m, L_m) then
14       count ← count + 1
15       if count ≥ EarlyPruneCount then
16         continue
17   return arg max_{p∈𝒫,c} U(p, c)
```

processing time (i.e., time spent in a filtering module or GPU inference time)[2] (Algorithm 3 lines 10-11). If a chosen pipeline configuration c has been explored by a previous placement choice, Vulcan calculates the utility for the new placement p by estimating the new query end-to-end latency $L_{p,c}$ and resource consumption $R_{p,c}$ without launching the pipeline (lines 7-8,12). $L_{p,c}$ is estimated by summing up the total processing time of each operator measured offline excluding the DNN module, the additional network latency introduced by placement, and the updated GPU inference latency as shown below:

$$L_{p,c} = L_{offline,total} - L_{offline,gpu} + \sum L_{p,net} + L_{p,gpu} \tag{4.5}$$

The network component of the new latency, $\sum L_{p,net}$, is calculated by summing up the network latency going across two adjacent tiers. Figure 4.7 illustrates how this process works. The latency is calculated by taking the ratio of a component's output size (cached per configuration) to the assigned link bandwidth capacity the query data traverse through. Note that only the components sending data to the next tier in the infrastructure are considered. The GPU inference latency, L_{gpu},

[2] Vulcan also collects the size and bandwidth of the data source to account for the case where only the data source is placed on the first tier.

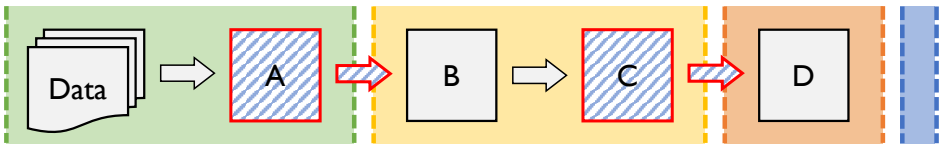


Figure 4.7: An example of how additional network latency is calculated for a pipeline with 4 operators placed across the edge infrastructure (i.e., device edge → on-premise edge → public MEC → cloud). Output sizes of shaded operators are used to calculate the additional latency introduced by the placement.

is updated by multiplying with a coefficient based on the GPU type to reflect the performance difference, which we determined by profiling all GPUs available in our cluster. $R_{p,c}$ is estimated in a similar way by including additional network bandwidth and GPU processing time introduced by the placement. After all placement choices are explored, Vulcan selects the placement (together with the optimal configuration) that achieves the highest utility (line 17).

4.4.3.2 Pruning Unpromising Placement Choices

Although caching intermediate results allows Vulcan to never re-launch a pipeline with the same configuration, exhaustively exploring all feasible placement choices may still incur large search cost as the profiler strives to find a good configuration for an unpromising placement. For example, a query preferring latency performance will not favor the placement that places the filtering module too far away from the data source.

To apply early pruning, we set a utility threshold which is equal to the utility value when both minimum performance metrics is used for the given query configuration and placement, namely $U(Q_m, L_m)$, which is evaluated to zero in our utility definition (§4.4.1). When the profiler obtains N utility values below the threshold, we early prune the current placement choice (Algorithm 3 lines 13-16). Compared to other alternative pruning solutions, such as building a statistical model for pruning decisions, this simple scheme works well because Vulcan profiler is designed to always pick more promising configuration than its last attempt (§4.4.4), and consecutive bad utility values indicate a high probability of unpromising placement. This also allows us to set N to a small value ($N = 3$ in our implementation) as otherwise the profiler would have already terminated itself after a small number of attempts.

Placing split ML models. Recent research [74] has proposed the idea of splitting the layers of a large DNN model and placing them at different edge tiers. Vulcan can handle such design as long as the split layers are properly specified in the input queries as individual modules.

4.4.4 Determining Query Configuration

For each placement choice, Vulcan leverage Bayesian Optimization (BO) [158, 43] to efficiently explore pipeline configurations that achieve the best performance with minimized resource consumption (see Figure 4.5).

4.4.4.1 Why Do We Choose BO?

BO is a methodology for optimizing expensive objective functions and is widely applied to many computer systems [202, 141, 149, 23, 42, 122]. At a high level, it learns the shape of the objective function by picking inputs that has the highest probability of reaching the global maximum of the function. As BO accumulates more observations, it becomes more confident on the actual shape of the objective function. This is a good fit for our solution as we need to quickly find the optimal configuration that maximizes the utility function without exploring too many choices.

More importantly, we observe BO has many advantages over other popular optimization schemes in the context of live ML queries. Greedy Hill Climbing has been applied in query configuration for video analytics queries [226], but it is purely exploitative and cannot perform as efficient global exploration as BO does (§4.6.3 provides detailed evaluation). Multi-Armed Bandit (MAB) [95] is another popular optimization scheme for sequential decision making, but it is designed for optimizing cumulative rewards, contrasting our goal of finding the one configuration with the highest utility. In addition, BO tunes the entire set of input configurations all together for each iteration no matter how large the input vector is, whereas MAB can only adjust one knob at a time. We also prefer BO over population-based optimization scheme such as Particle Swarm Optimization (PSO), as applying PSO in query configuration requires launching many ML pipelines in parallel, leading to a much high computation cost (e.g., GPU resourse).

4.4.4.2 Applying BO to Query Configuration

We define the objective function of BO as $f(\vec{x})$, which models how good a given query plan is based on a given pipeline and a physical placement choice. The input $\vec{x}$ is the set of query configuration knobs, and the output of f is the utility value, $U_{q,p,c}$ for a given pipeline q with placement p and a set of configurations c. For each iteration, Vulcan launches the pipeline with the configurations suggested by BO (i.e., $\vec{x}$), and collects the measurements to compute $U_{q,p,c}$, which is then fed back to BO as the new observation.

Choice of prior and acquisition functions. Internally, BO learns an objective function by leveraging a *prior function* and an *acquisition function*. The former represents the belief about the space of possible objective functions, whereas the latter guides BO to choose the next promising input where the value of acquisition function is maximized. We choose Gaussian Process as the

prior function and use *Matern 5/2* as its covariance function to describe the smoothness of the prior distribution, which is known to perform well among systems applying BO [202, 23]. Among three major choices of the acquisition function, namely probability of improvement [129] (PI), expected improvement [109] (EI), and upper confidence bound [204] (UCB), we select UCB as it works the best for our workloads. We defer a dynamic approach of selecting acquisition functions [44] to future work.

Starting and stopping BO. We start with N random sets of input query configurations as initial observations for BO to learn the rough shape of the objective function. We set $N = 3$ in all of our experiments and found it works well for various workload settings. Vulcan stops BO when the improvement of the utility value is less than a threshold for a few consecutive runs (i.e., 10% for 5 consecutive runs, which empirically works well). We include a sensitivity analysis on how the parameters we use in the starting and stopping conditions affect BO's performance in Section 4.6.7.

One alternative design which seems promising but we do not consider is to directly apply BO for placement selection and pipeline construction by treating available placement choices and pipelines as another two knobs in the total configuration space. In BO, input parameters are specified in a continuous range. For example, the voxel size in the voxelization module of an AD perception pipelines is chosen between 0.1 and 0.5. As BO moves within this range, the behavior of the module changes with the size of the voxel, leading to a smoother shape of the objective function with more predictable outputs. In contrast, the behavior of different placement choices or filter orderings is hard to predict, and going through the configuration space of those (i.e., placement or pipeline choices indexed by numbers) leads to very rough shape of the objective function for BO to grasp; thus it becomes much more difficult to find the global optimum.

4.5 Online Adaptation

This section describes how Vulcan performs online adaptation to handle runtime dynamics after query deployment. Vulcan currently does not support concurrent execution of different ML queries, and we defer the support of multi-resource sharing among concurrent live ML pipelines to future work.

4.5.1 Detecting and Handling Runtime Dynamics

Vulcan leverages two design ideas to quickly converge back to the best query plan during online adaption: (i) monitor utility change to detect runtime dynamics, and (ii) leverage prior knowledge during reprofiling.

Detecting runtime dynamics. Vulcan detects runtime dynamics by monitoring the change in a

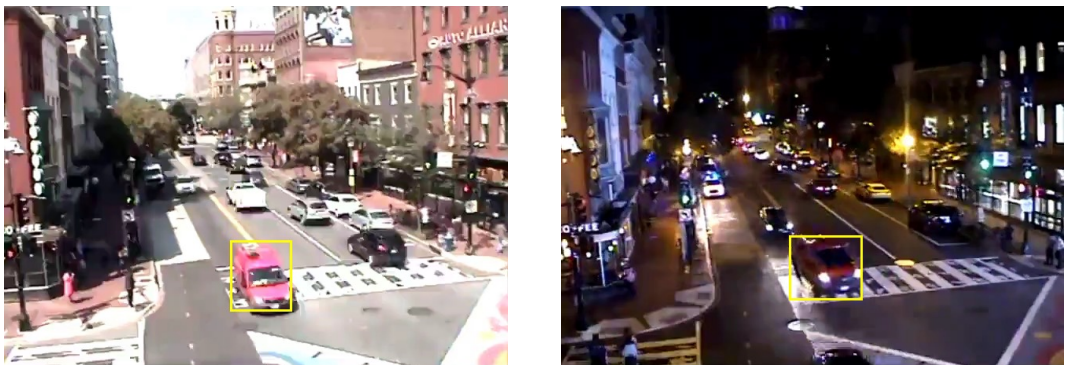


Figure 4.8: Scenes taken during different time of day from a video query detecting red vehicles.

query's utility values, as any runtime dynamics (content, network, or compute based) leads to a change in utility. This includes all the measurements required to calculate up-to-date utility (i.e., latency, accuracy, module output size and bandwidth), and they are periodically reported to Vulcan monitor via HTTP requests. To obtain real-time ground truth on query accuracy given unlabeled live data, Vulcan launches a duplicated pipeline with the most expensive configuration inside the cloud, which only receives live data periodically to minimize network cost. A substantial change in the utility triggers reprofiling which deploys the query in the cloud similar to the case of offline profiling. We set the threshold of utility change empirically (10% in our implementation) via profiling.

Leveraging prior knowledge in reprofiling. We observe that most of the query, such as the object of interest and where the query takes place, remains the same after deployment; meaning, we can take the advantage of prior knowledge from offline profiling. Figure 4.8 shows two example scenes taken by the same camera during different time of the day from a query that detects red vehicles. We observe a high level of similarity between the two scenes except for environment illumination. Let us define the distance between two configurations, C_A and C_B, to be the total number of steps needed for each configuration knob in C_A to change to each knob value in C_B. In this example, applying the same configuration from daytime to nighttime scenes leads to an average 26.2% utility drop among all placement choices, but it requires only an average distance of 2.47 steps to converge back to the query plan with the highest utility (figures omitted in the interest of space).

To apply prior knowledge, Vulcan makes the following changes to the normal profiling process. Vulcan keeps track of the most recent top-K and worst-K configuration per placement choices ($K = 3$), and applies them as initial data points in BO such that BO can quickly grasp the shape of the objective function. We also tune up the exploitation factor in the acquisition function (κ in UCB [204]) to $1/10$ of its original value to focus more on exploitation than exploration. Note that Vulcan does not just stick to the best placement choice but choose to re-perform placement

```
HttpProcessor _httpProcessor = new HttpProcessor(Config.ENDPOINT_URL_SAMPLING);
Int sampling_rate = 3; // selecting 1 out of 3 frames
var form = new MultipartFormDataContent();
form.Add(new StringContent(sampling_rate.ToString()), "frameSamplingRate");
var result = await _httpProcessor.PostAsync(form);
```

(a) Sending Configuration Updates

```
// POST: api/Config
[Produces("application/json")]
[HttpPost]
public async Task<IActionResult> PostConfig([FromQuery] string frameSamplingRate = null)
{
  if (frameSamplingRate != null)
  {
    Config.FRAME_SAMPLING_RATE = Int32.Parse(frameSamplingRate);
    /* sampling rate updated for subsequent frames */
  }
  return Ok();
}
```

(b) Posting Configuration Updates

Figure 4.9: Code snippets of Vulcan APIs on dynamically updating query configuration. (a) Vulcan Profiler sending the configuration updates to deal with runtime dynamics. (b) Vulcan Controller at the container updates the configuration to use the updated value.

selection. As runtime dynamics can involve network and compute resource changes, this allows Vulcan to change the placement of the pipeline when merely adjusting query configuration makes little impact on recovering query performance. We evaluate how Vulcan performs under different types of runtime dynamics in Section 4.6.6.

4.5.2 Enabling Online Adaptation

After identifying the right query plan upon runtime dynamics, we must also enforce this at production scale. Unfortunately, existing frameworks for large-scale deployment, such as Kubernetes [13], do not support container modification during execution. Therefore, Vulcan adds its own implementation.

Dynamically updating query configuration. Figure 4.9 shows the example interfaces of Vulcan sending configuration updates and posting them inside the container upon a change in the frame sampling rate of a traffic monitoring query. In Vulcan, each pipeline module is installed and launched by a container. Query configurations are updated in real time without stopping the containers. If any configuration knobs need to be updated after reprofiling, Vulcan sends out updated configuration using HTTP requests to the containers that launch the corresponding modules (Figure 4.9a). The exact location (ENDPOINT_URL_SAMPLING) of the containers is acquired

when Vulcan first deploys the pipeline. The containers receiving the request update the configuration right away, without stopping the current ML task (Figure 4.9b).

Handling placement changes. If the new query plan involves a placement change (e.g., during network or compute resource change), Vulcan migrates the container of the corresponding module to the updated tier. To perform the migration, Vulcan first launches the module with the same query configuration on the target tier. It then updates the location of the new container to the upstream module via the same API for configuration updates described in §4.5.1, before removing the old container on the current tier. The new location of the container is also updated in Vulcan monitor for future adaptation.

4.6 Evaluation

We evaluate the effectiveness of Vulcan in performing automatic query planing in terms of profiling time, query performance, and query resource consumption. Our key findings:

(1) Vulcan generates query plan with better profiling cost by 4.1×-30.1× over state-of-the-art ML analytics systems while delivering up to 2.0×-3.3× better latency (§4.6.2).

(2) Vulcan performs better query configuraiton, placement selection, and pipeline construction by outperforming existing solutions with up to 2.8× better query latency and 174× lower network resource consumption (§4.6.3-§4.6.5).

(3) Vulcan achieves consistently better 99^{th}-p latency performance (by up to 2.5×) during online adaptation over the-state-of-the-art (§4.6.6).

4.6.1 Experiment Setup

Live ML Queries. We illustrate Vulcan's performance in performing query planning for three example live ML queries:

- *Video Monitoring*: monitors the traffic volume by examining live video frames and counting the vehicles of a specific color (white color in our examples).
- *Autonomous Driving Perception*: takes 3D point cloud as input and generates a real-time perception (represented as 3D bounding boxes) surrounding a vehicle.
- *Automatic Speech Recognition*: converts live human speech (with background noises) into written text.

Datasets. We adopt several real-world datasets for each query example to perform a comprehensive evaluation. The video monitoring queries use videos captured by traffic cameras among different metropolitan areas in Bellevue and Washington D.C. The autonomous driving queries use

Table 4.1: ML model variations used in evaluation.

Model	# Parameters (Million)				
YOLO [185]	**v5n6**	**v5s6**	**v5m6**	**v5l6**	**v5x6**
	3.2	12.6	35.7	76.8	140.7
PointPillars [130]	**SECFPN**		**SECFPN (FP16)**		
	4.9		4.9		
SSN [236]	**SECOND**		**RegNet**		
	6.2		7.1		
CenterPoint [223]	**DCN**		**Circular NMS**		
	6.0		6.1		
wav2vec2 [32]	**base**	**large_10m**	**large_960h**		
	94.4	315.5	315.5		
HuBERT [224]	**large**		**xlarge**		
	315.5		962.5		

LiDAR sensor data from nuScenes [45]. Automatic speech recognition queries use the VOiCES dataset [186] with background noise enabled. Appendix C.1.1 has more details.

Pipelines. Unless otherwise specified, video monitoring queries in our evaluation experiments use the same pipeline which consists of the following components in order: a background subtractor to detect moving vehicles, a color filter, and a variations of YOLOv5 [11] object detector. The autonomous driving pipeline feeds 3D point clouds into a ground removal module, followed by a voxelization module and a variations of PointPillars [130], SSN [236], or CenterPoints [223] 3D object detector. The speech recognition queries use a pipeline that consists of an audio sampler, a noise reduction module, a variation of wav2vec2 [32] or HuBERT [224] model, and a decoder. Table 4.1 records all ML model variations we use in the evaluation. Appendix C.1.1 describes the details of the query configuration knobs for all queries.

Baselines. We consider the following baselines for Vulcan.

• *Exhaustive search.* To determine the optimal placement and configuration for a ML pipeline, we use exhaustive search to explore all possible placement choices and configurations.

• *ML analytics systems.* We implement four state-of-the-art ML analytics systems: VideoStorm [226], Chameleon [107], JellyBean [221], and LLAMA [189]. Both VideoStorm and Chameleon adopt a variant of greedy-hill climbing during query profiling, and Chameleon applies spatial and temporal correlations to reduce the search cost during online adaptation. JellyBean selects models with target accuracy and lowest cost, and applies beam search to determine query placement in a greedy fashion. LLAMA dynamically explores query configuration by computing per-invocation pipeline latency.

• *Pipeline placement strategies.* Besides JellyBean's greedy placement, we implement the following commonly-adopted placement strategies: (1) *prioritizing network (PN)* which places operators closer to the device edge, (2) *prioritizing compute (PC)* which places operators closer to the cloud,

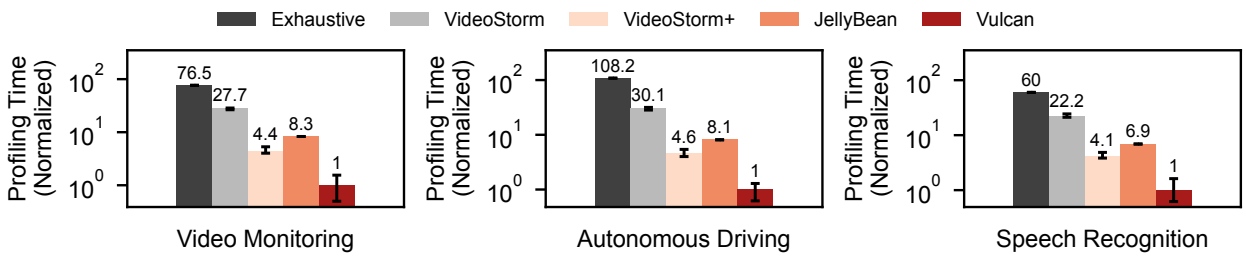


Figure 4.10: Comparing the profiling cost of video monitoring, autonomous driving, and speech recognition queries.

and (3) *balancing network and compute (NC)* which places filtering modules closer to the device edge and the remaining operators to the cloud.

Emulation Setup. We emulate a setup of 4 edge tiers consisting of the device edge, the on-prem edge, the public MEC, and the cloud, which is consistent with our production edge infrastructure by adding additional network latency and compute overhead. The network bandwidth and compute cost are emulated based on the real network and hardware settings in our production infrastructure.

We run all experiments 20 times with different random seeds, and collect 10th, 50th, and 90th percentiles of data to include the effect of randomness in Vulcan and other baselines. The 10th and 90th percentiles are plotted via error bars unless otherwise specified. We set L_m for video monitoring, autonomous driving, and speech recognition queries as 2000ms, 400ms, and 500ms, respectively. Q_m is set at 0.8 $\times$ the accuracy achieved by the most expensive configuration for each type of queries. We set γ (preference of query accuracy over latency) to be 0.5 for all queries unless otherwise specified.

4.6.2 End-to-End Improvement

We start with showing the end-to-end improvement of Vulcan over other baselines in generating query plans for all three types of queries. We compare exhaustive search, original VideoStorm that explores all placement choices, VideoStorm+ that explores a combination of all three baseline placement strategies (i.e., PN, PC and NC) on top of VideoStorm, and JellyBean. We compare Vulcan with Chameleon and LLAMA during online adaptation in §4.6.6 as their query configuration by design happens in the online phase.

We record profiling time normalized by the time spent by Vulcan, which achieves the smallest value in both types of queries. We also record performance (accuracy, median latency, and 99^{th}-p latency) and resource consumption (network and compute) achieved by the query plan and normalize the results based on the optimal query plan generated by exhaustive search. For VideoStorm, VideoStorm+, and JellyBean, we record the query plan achieved with highest utility

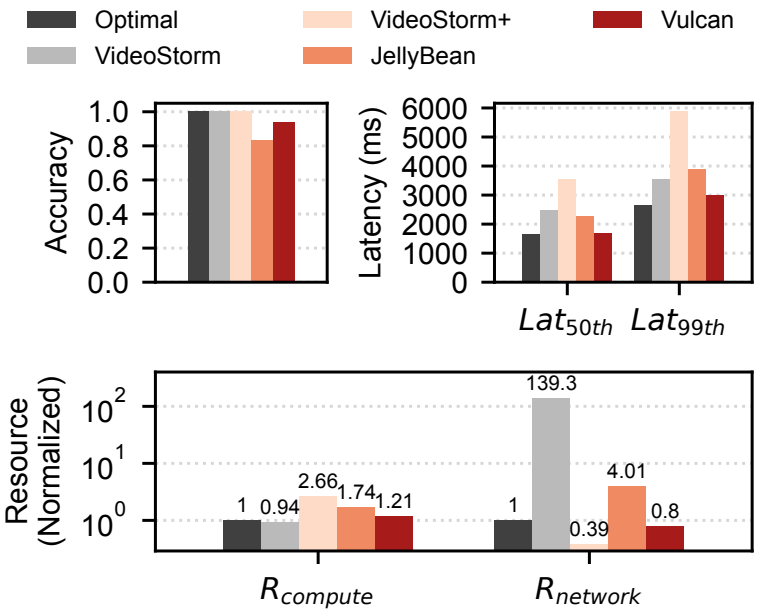


Figure 4.11: Comparing end-to-end performance and resource consumption for video monitoring queries.

given the same profiling time as Vulcan.

Figure 4.10 records the profiling cost for all types of queries, and Figure 4.11 summarizes the performance and resource consumption for video monitoring queries. Results for autonomous driving and speech recognition queries are similar and moved to Appendix C.1.2 in the interest of space. Vulcan achieves significantly lower profiling cost than exhaustive search, VideoStorm, VideoStorm+, and JellyBean by at least 60×, 22.2×, 4.1×, and 6.9× respectively across all three types of queries, and its query performance and resource consumption are very close to the optimal one achieved through exhaustive search, achieving up to 2.0×-3.3× better tail latency than other baseline approaches. Vulcan's performance improvement comes from its joint optimization of pipeline placement and configuration with low cost. Vulcan avoids exhaustively searching for optimal placement by reusing pipeline results. VideoStorm+ improves profiling time by only exploring a few placement choices, at the cost of worse latency and network resource consumption, and it is still outperformed by Vulcan by at least 4.4×. JellyBean's placement algorithm greedily finds promising placement choices but separately optimizes query configuration and placement, leading to sub-optimal latency, accuracy, and resource usage.

4.6.3 Selecting Better Query Configurations

We next delve into the performance of each profiling components in Vulcan, starting with query configuration. Figure 4.12 shows the profiling cost among Vulcan and other baselines with the same

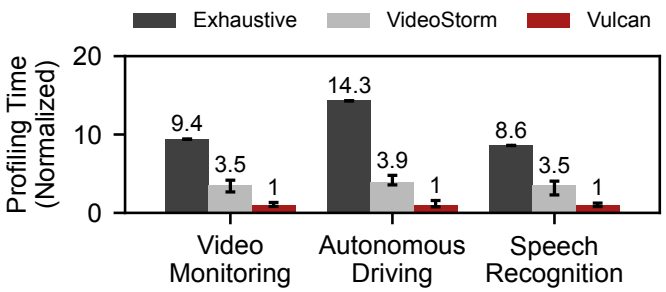


Figure 4.12: Profiling cost of query configuration given the same pipeline and placement.

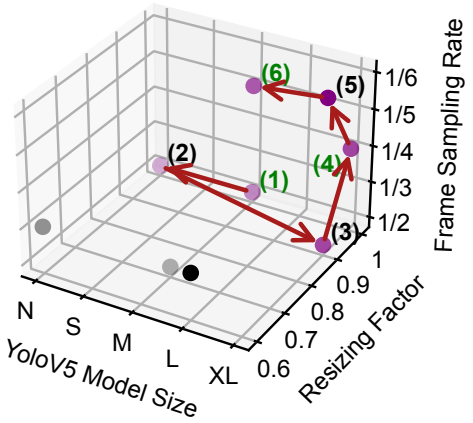


Figure 4.13: BO's search path in selecting configurations. The new maximum in utility is marked in green. The initial random configurations are shown in black.

pipeline and the best placement choice. Even without the benefits achieved in efficient placement selection, Vulcan can still outperform VideoStorm, which leverages greedy hill climbing, by 3.5× in profiling time. To illustrate how Vulcan achieves this, we plot one of Vulcan's example BO search path in the video monitoring query in Figure 4.13. BO focuses its exploration on larger resizing factor after verifying that smaller values lead to worse performance, and finds the optimal query configuration at the 6^{th} step. BO takes 5 more steps to confirm we are not likely to encounter better results before stopping, which we do not plot for legibility.

4.6.4 Selecting Better Placement

We evaluate Vulcan's placement decisions by comparing with common placement strategies (§4.6.1) and JellyBean. Each baseline placement strategy uses exhaustive search to find the optimal pipeline configuration, and all comparisons use the same pipeline. We ignore the profiling time improvement of Vulcan over other baselines and only focus on comparing the achieved performance and resource consumption. Figure 4.14 illustrates the results of video monitoring queries. Results for

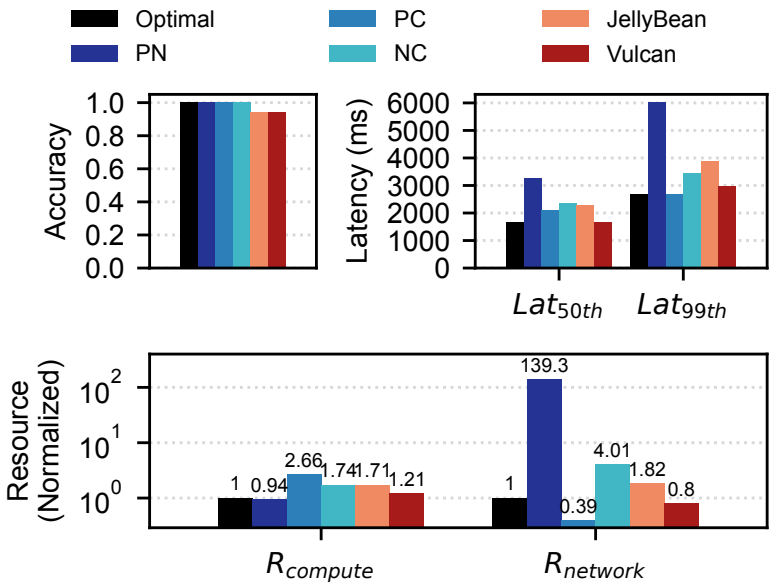


Figure 4.14: Comparing Vulcan with different placement strategies on serving video monitoring queries.

autonomous driving and speech recognition queries are similar and shown in Appendix C.1.3. We observe that PN prioritizes network latency but fails to consider the large ML inference latency on slower compute nodes, resulting in worse end-to-end query latency. PC optimizes ML inference latency, which does not always lead to better latency. Moreover, it consumes significantly more network resources. NC tries to strike a balance between PN and PC but still achieves worse latency. JellyBean's placement algorithm is based on a fixed query configuration that is determined in a prior stage, leading to a sub-optimal placement choice. In comparison, Vulcan always achieves the same placement choices as the exhaustive search and delivers up to 2.8× better query latency and 174× network resource consumption than other baselines, thanks to its joint optimization between placement and configuration. The performance gap between Vulcan's query plan and the optimal query plan is caused by Vulcan picking a different pipeline configuration, in which case Vulcan takes much less profiling time with similar performance and resource consumption.

4.6.5 Selecting Better Pipelines

We now evaluate how well Vulcan selects the filter ordering during pipeline construction based on performance requirements of the queries. We compare the performance of fixed pipelines with Vulcan's choices, which is dynamically determined based on F_γ in Eq 4.4 (§4.4.2). To focus on whether Vulcan makes the correct choice of filter ordering, we remove the potential noise of BO by using the best pipeline configuration determined by exhaustive search. We sweep γ and record

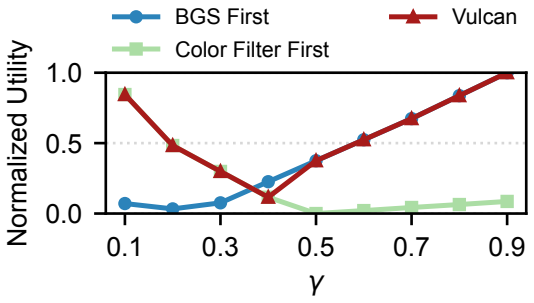


Figure 4.15: Compare Vulcan's selection of filter ordering with fixed pipeline settings in video monitoring queries.

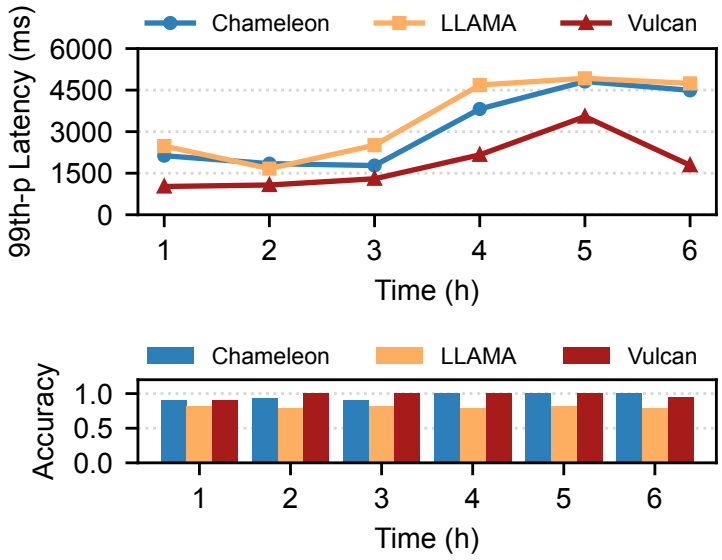
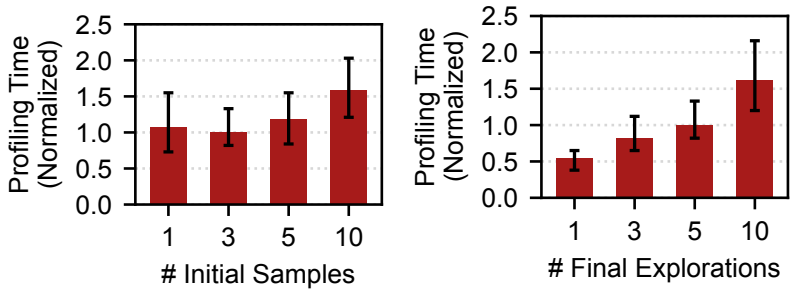


Figure 4.16: Comparing Vulcan with Chameleon during online adaptation for video monitoring queries.

the utility value achieved by picking the corresponding pipeline. Figure 4.15 shows that Vulcan is able to select the best pipeline in 8 out of 9 cases. On the other hand, sticking to the other two fixed pipeline settings leads to only 6 out of 9 cases and 3 out of 9 cases for selecting the correct filter ordering.

4.6.6 Handling Runtime Dynamics

To evaluate Vulcan's online adaptation, we compare it with Chameleon and LLAMA, the state-of-the-art solution in adapting runtime dynamics in ML queries. We took a continuously running 6-hour long video from our Washington D.C. video dataset to monitor red vehicles. Performance (99^{th}-p latency and accuracy) was collected and reported at the end of every hour. All three systems



(a) Starting Condition (b) Stopping Condition

Figure 4.17: Sensitivity analysis of BO's parameters.

started with the same video monitoring pipeline and placement, where the background subtractor and the color filter are placed on the device edge, and the object detector is placed on the public MEC. At the end of the 3rd hour, we reduced the link capacity from the on-premise edge to the Public MEC by 20× to simulate a network outage. We see in Figure 4.16 that Vulcan achieves consistently better latency than Chameleon and LLAMA by up to 2.5×. Neither Chameleon nor LLAMA updates pipeline placement upon resource changes, causing worse latency after the outage. On the other hand, Vulcan adjusted the pipeline placement by placing the object detector onto the on-premise edge after the outage, thus mitigating the latency hikes.

4.6.7 Sensitivity Analysis

We analyze the sensitivity of two parameters used in BO during pipeline configuration in Figure 4.17. We use the same experiment setting in Figure 4.12 with video monitoring queries. Figure 4.17a plots the profiling time normalized by the value picked in Vulcan implementation when sweeping the number of initial random configurations passed to BO. The number of random configurations does affect the profiling time but has a negligible impact on the quality of the query plan achieved. Figure 4.17b sweeps the stopping threshold of BO, which is the number of consecutive rounds without significant improvement. BO needs at least 3 rounds to obtain a good chance of finding optimal configurations, but adding more rounds won't further improve the quality of profiling.

4.7 Related Work

Configuration management for ML analytics. Finding better configurations for ML analytics is a well studied research topic [226, 107]. VideoStorm [226] investigates query's accuracy-

latency profile and applies a greedy hill-climbing approach to search for query configurations. Chameleon [107] applies spatial and temporal correlations of video frames to reduce the search cost of query configurations during online adaptation. Although VideoStorm and Chameleon works well for ML quries with fixed pipeline placement, Vulcan jointly explores placement and configurations for live ML queries deployed across a heterogeneous infrastructure.

Constructing ML pipelines with declarative queries. Researchers have proposed declarative query languages, especially for video analytics, to construct ML pipelines [37, 63, 144, 188]. MIRIS [37] provides a declarative interface to select video object tracks and selects corresponding modules based on user specification. Viva [188]'s declarative interface allows users to specify relational hints to express the relationship among different modules, which helps Viva to construct and optimize the pipeline. However, those works only consider accuracy requirement when optimizing pipelines and ignore the resource demands especially network usage, providing no guarantees to end-to-end latency performance.

Domain-specific optimization for ML analytics Applying domain-specific knowledge to perform optimization for ML analytics is an active research area spanning many use cases including video analytics [63, 107, 37], autonomous driving perception [227, 182, 228], and automatic speech recognition [139, 64]. BlazeIt [63] leverages spatiotemporal information of objects in video to optimize aggregation and limit queries. VI-Eye [228] achieves better accuracy and latency performance by exploiting domian knowledge in autonomous driving scenarios to recognize key semantic objects that can be used to align vehicle-infrastructure point cloud pairs. Vulcan's design is orthogonal to domain-specific techniques and can be applied to different use cases without additional changes.

Continuous learning for ML analytics. Another line of work that gets increasingly more attention nowadays is continuous learning in ML analytics [39, 119, 146]. Ekya [39] jointly schedules and allocate resources to ML retraining and inference to handle data drift. RECL [119] integrates model reusing with model retraining to quickly adapt to a lightweight expert DNN model for each specific video scenes. Techniques leveraging continuous learning is complementary to Vulcan and can be applied to Vulcan's online adapation phase. On the other hand, Vulcan's design can be applied to those works as well, including dynamically updating query configurations and fast detection of data changes by monitoring utility changes.

Reducing cost of ML analytics via filtering. There have been recent studies on applying different types of filtering techniques to reduce the resource consumption of ML analytics without compromising accuracy [114, 98, 147, 94, 47]. NoScope [114] searches for and trains a cascade of models that preserves the accuracy of the ML inference but with far less computation cost. Focus [98] uses cheap convolutional network classifiers (CNNs) to construct an approximate index of all possible

object classes in the video frame, which reduces the use of expensive CNNs during query time. Probabilistic Predicates [147] constructs and applies binary classifiers to filter out data blob that will not pass query predicate and thus accelerate queries with expensive user-defined functions. Vulcan builds up on the idea of applying filtering and propose a novel technique to order the filters for ML pipelines.

4.8 Conclusion

Serving live ML analytics involves constructing, placing, and configuring ML pipelines as well as their online adaptation. However, existing query planning solutions for live ML queries remain elusive with piecemeal and sub-optimal. We present Vulcan, an ML analytics system that performs automatic query planning for live ML analytics. Vulcan automatically construct the pipeline and determine the best ordering of filtering operators for query performance. It efficiently explores placement choices by reusing of intermediate pipeline profiling results, and leverage Bayesian optimization with prior knowledge to handle query configuration and online adaptation. Vulcan outperforms state-of-the-art solutions on profiling time, query latency, and resource consumption when serving queries with real-world datasets.

CHAPTER 5

Mercury: QoS-Aware Tiered Memory System

Finally, we move our research interests back to the datacenter and study QoS in computer memory. Tiered memory systems, especially those enabled by CXL, have become a popular choice to increase memory capacity to satisfy the increasing memory demand in cloud applications. Compared to traditional single-tier memory systems, tiered memory systems can accomodate more applications without losing much performance, as long as applications' hot memory can be quickly identified and moved to the fast memory tier (e.g., local DRAM) in the system. In this chapter, we introduce Mercury, a QoS-aware tiered memory system we build to provide better QoS guarantees for memory-intensive applications.

The remaining of this chapter is organized as follows. Chapter 5.1 introduces Mercury. Chapter 5.2 explains why the existing tiered memory systems lack QoS support. In Chapter 5.3, we perform a detailed analysis on two sources of performance unpredictability in tiered memory systems. Chapter 5.4 and Chapter 5.5 describe the system overview and design details of Mercury. Our evaluation results are discussed in Chapter 5.6, followed by a discussion of related work in Chapter 5.7. We finally conclude Mercury in Chapter 5.8.

5.1 Introduction

With the increasing memory demands of datacenter applications, tiered memory systems have widely been adopted to replace DRAM-only systems [151, 184, 222, 72, 194, 133]. Many recent tiered memory systems are enabled by Compute Express Link (CXL) [15], which is a high-speed interconnect standard that allows memory capacity to grow by attaching more memory to the CPU, without losing nanosecond-scale memory access latency. The increased memory capacity enables deployments of more memory-intensive applications that fall into two broad categories: (1) *latency-sensitive (LS)* applications such as in-memory key-value store [17, 18] that require low-latency memory access, and (2) *bandwidth-intensive (BI)* applications such as large-memory ML models (e.g., long-context language models or recommendation models [163, 189]) that require a

sustained memory bandwidth.

In production environments, running a single application on a tiered memory system wastes both memory capacity and bandwidth. As a result, multiple memory-intensive applications often have to share the tiered memory system to maximize resource utilization and improve total cost of ownership (TCO) [151, 72, 133, 194, 166].

Existing research on tiered memory systems primarily focuses on page temperature monitoring and efficient page migration to better utilize local memory resources (i.e., fast-tier DRAM) [184, 222, 121, 151, 20, 116]. However, these solutions optimize a single application running on a single server. They are not built with Quality-of-Service (QoS) support and thus cannot react to applications with different service level objectives (SLOs). In particular, existing tiered memory systems suffer from great performance unpredictability due to two reasons. First, multiple applications can *contend for local memory* (i.e., fast-tier memory), and the ones with hotter memory get more resources. As a result, a low-priority application may grab more local memory than a high-priority application, leading to *priority inversion* (§5.2.1). Second, high memory bandwidth generated by BI applications can hurt the performance of coexisting LS applications on the same tier or even across tiers – we refer to these phenomena as *intra- and inter-tier interference*, respectively. To the best of our knowledge, no existing solutions have systematically tackled bandwidth interference across tiered memory, including the recent proposal on QoS support for tiered memory [72].

In this chapter, we aim to provide predictable performance for applications with different SLOs in the presence of local memory contention and memory bandwidth interference. However, achieving this goal faces the following unique challenges. First, it requires an efficient way to track and control memory resources on each memory tier, but the memory control mechanisms in existing operating systems (OSes) are not designed for tiered memory. Second, it is non-trivial to determine the amount of memory to allocate to each application in each tier while simultaneously maximizing resource utilization to accommodate more applications. For example, in the case of a BI application with a loose SLO, although migrating some of its local memory to CXL memory can save local memory for another LS application, too much migration may incur inter-tier interference, leading to SLO violations. Finally, an application's workload can change after its deployment, calling for an efficient approach toward performance monitoring and real-time adaptation.

We present Mercury, a QoS-aware tiered memory system that provides predictable performance for memory-intensive applications. Mercury incorporates the following key ideas to overcome the aforementioned challenges:

(1) Mercury enforces application-level resource management by enabling *per-tier page reclamation*. This allows Mercury to control an application's available local memory while still leveraging existing page migration designs.

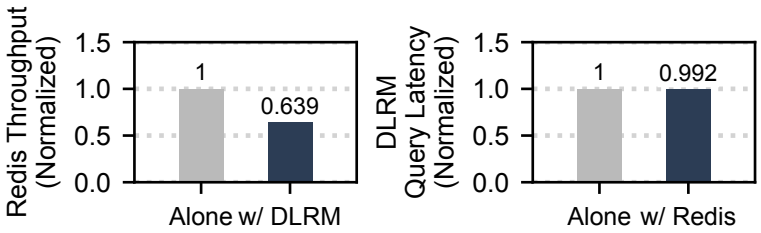


Figure 5.1: Unpredictable performance of Redis and DLRM as they compete for local memory on the fast tier. Existing solutions cannot distinguish among applications when migrating their hot pages, and thus cannot provide QoS guarantees.

(2) Mercury includes a novel *admission control* algorithm to determine the right amount of resources for applications to satisfy SLOs with the goal of maximizing local memory utilization while mitigating memory bandwidth interference.

(3) Mercury provides real-time adaptation to unpredictable memory and bandwidth changes, preventing sudden system burdens while maximizing the number of applications meeting their SLOs based on priority.

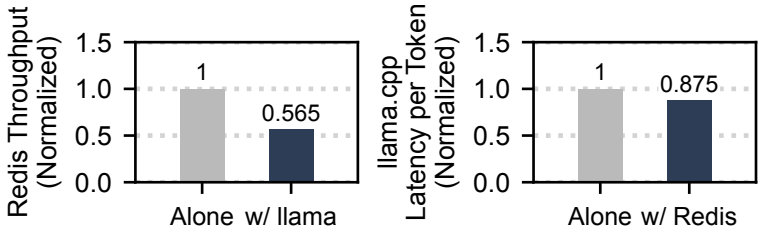
We evaluate Mercury's effectiveness in providing QoS using real-world applications and find that Mercury can closely track SLOs at different memory access latency and bandwidth targets. More importantly, Mercury is able to handle local memory contention as well as memory bandwidth interference at various multi-tenant settings, achieving up to 53.4% better application performance than TPP while satisfying more applications' SLO. Mercury also achieves 8.4× longer SLO satisfaction time than TPP in a long-running experiment to handle real-time workload changes.

We make the following research contributions:

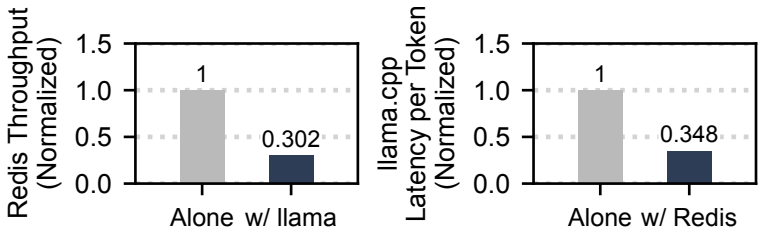
- We conduct a thorough QoS analysis on production-ready tiered memory systems, including local memory contention and memory bandwidth interference, and draw new observations.
- We propose a novel admission control and real-time adaptation algorithm tailored for tiered memory systems to achieve different SLOs for coexisting applications.
- We implement a new kernel-level resource management scheme to control resources on tiered memory.

5.2 Lack of QoS Support in Tiered Memory

We start by introducing why existing multi-tiered memory systems lack QoS support. There are two root causes for unpredictable performance when multiple applications coexist – local memory contention (§5.2.1) and memory bandwidth interference (§5.2.2).



(a) Intra-tier Interference



(b) Inter-tier Interference

Figure 5.2: llama.cpp's memory bandwidth creates interference, resulting in a significant drop in the throughput performance of Redis. (a) shows intra-tier interference when llama.cpp is on the same fast tier with Redis. (b) shows inter-tier interference after all llama.cpp's memory is migrated to CXL.

Characterizing applications. In this work, we classify applications into two types: (1) *Latency-sensitive (LS)* applications that desire low memory access latency, and (2) *bandwidth-intensive (BI)* applications that require sustained memory bandwidth. Both types of applications can be deployed onto the same server to increase system-level memory utilization. We do not assume LS applications are always more important than BI applications, and vice versa.

Hardware Specifications. Our motivating experiments run on dual-socket AMD Genoa servers each having 96 physical CPU cores and 12 channels of DDR5 memory per socket. CXL memory is enabled via memory expansion cards with four channels of DDR4 memory. The total memory footprint of a single server is 1.8TB; each socket has 768GB DDR5 and CXL memory has 256GB DDR4 memory.

5.2.1 Local Memory Contention

The core design principle of multi-tiered memory systems is to keep hot pages on local memory (i.e., DRAM on the fast tier) while migrating cold pages to the slower tier [184, 222, 121, 151, 20, 116, 72]. However, as multiple applications have to share tiered memory to fully utilize the local memory and bandwidth capacity, such a design does not distinguish among applications with different SLO targets, and thus can not provide QoS guarantees. For example, a low-priority application with a

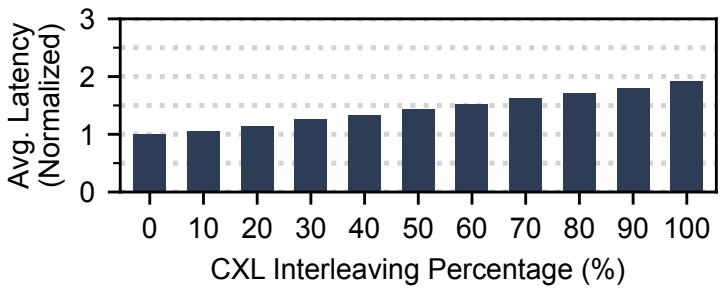
large amount of hot memory can still compete for local memory, hurting the performance of more important applications. Figure 5.1 illustrates this issue with two real-world applications, Redis [18] and Deep Learning Recommendation Model (DLRM) [163] on a real CXL-based tiered memory setup (without GPUs) used in a production-ready cluster. We enable TPP [151], the state-of-the-art multi-tiered memory system, to handle the page migration mechanism between local memory (the fast tier) and CXL memory (the slow tier). We configure the applications such that the sum of their working set size (WSS) is greater than the available local memory on the fast tier. The throughput of Redis drops by 36% due to insufficient local memory, as DLRM is competing for local memory at the same time. An ideal QoS-aware memory system should be able to *allocate the right amount of local memory* to each of the applications based on their QoS target.

5.2.2 Memory Bandwidth Interference

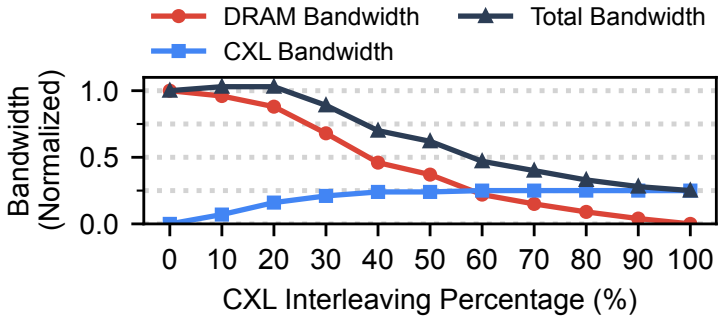
Another source of performance unpredictability is memory bandwidth interference, where applications with high memory bandwidth affect the performance of coexisting latency-sensitive applications. The problem is further complicated in tiered memory systems, where applications can generate bandwidth with memory requests accessing different memory tiers. We identify two types of memory bandwidth interference – **intra-tier interference** and **inter-tier interference**. Intra-tier interference refers to the one happening on the same tier, which is also common in conventional, single-tier systems [165, 161, 142, 48]. Inter-tier interference occurs when excessive memory requests on one memory tier cause a slowdown of requests on another tier. Figure 5.2 illustrates both types of interference by colocating llama.cpp [16] inference tasks (without GPUs) and Redis. Redis's throughput degrades by 43.5% when llama.cpp is generating a high memory bandwidth on the same fast tier in Figure 5.2a. We then migrate all llama.cpp's memory to CXL memory and observe even worse Redis performance (dropped by 70%) due to inter-tier inference (Figure 5.2b).

Existing solutions on mitigating memory bandwidth interference work either (1) on the memory request level, such as memory request prioritization [165] or memory channel partitioning [161], or (2) on the CPU core level to throttle memory bandwidth from best-effort tasks [48, 142]. The request scheduling approaches assume a single-tier memory tier and do not work across multiple tiers. The CPU throttling approaches are agnostic to multiple memory tiers and thus cannot resolve inter-tier interference.

Mitigating interference plays an important role in providing SLO guarantees for LS applications. In the meantime, we also want to provide QoS for BI applications and meet their bandwidth SLO whenever possible. An ideal solution should mitigate interference by migrating pages off the tier experiencing interference while leveraging additional bandwidth in other tiers to satisfy SLOs



(a) Impact on Latency Performance



(b) Impact on Bandwidth Performance

Figure 5.3: Latency and bandwidth performance at different CXL interleaving ratios to illustrate the impact of local memory.

of bandwidth-intensive applications. However, such migration is not straightforward given the existence of inter-tier interference, as we will show in our detailed analysis in the following section.

5.3 QoS Analysis in Tiered Memory

In this section, we quantitatively analyze how local memory contention and memory bandwidth interference affect application performance and draw key insights to design Mercury. All results are collected on the same CXL-based tiered memory system as in Section 5.2.

5.3.1 Impact of Available Local Memory

We start by studying the impact of available local memory on application performance, as applications can easily run short of local memory during memory contention. We develop two microbenchmarks to represent LS and BI applications, which we denote as *LS* and *BI* in Section **??** for brevity. *LS* performs random memory access among a 4GB memory region. *BI* allocates enough CPU cores to generate memory bandwidth at maximum capacity on 128MB region per core. These microbenchmarks allow us to easily control the proportion of memory access on CXL

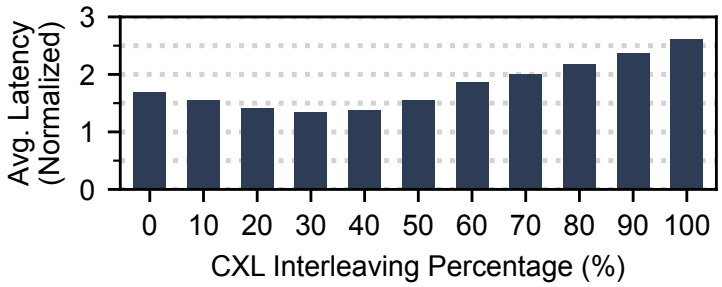


Figure 5.4: Performance of *LS* when *BI* is generating bandwidth at different CXL interleaving percentage. Migrating *BI* to CXL does not always lead to better performance of *LS* due to inter-tier interference. *BI*'s performance is very close to Figure 5.3b and omitted in the interest of space.

memory (the rest goes to local memory), which we denote as the CXL interleaving percentage. Figure 5.3 shows the impact of available local memory on latency and bandwidth performance by varying the CXL interleaving ratio of the two microbenchmarks. We observe the memory access latency of *LS* is proportional to its available local memory, and it becomes 2× slower when all memory is moved to CXL in the worst case. On the other hand, *BI*'s bandwidth performance degrades as more memory is accessed from CXL, dropping to 25% of its original performance when all memory is accessed via CXL.

Takeaway #1: Local memory directly affects the performance of both types of applications and should be allocated judiciously during memory contention.

5.3.2 Deep Dive in Memory Interference

We now take a close look at how LS applications are affected by memory bandwidth interference from BI applications.

Varying *BI* across tiers. In this experiment, we configure *LS* to always access local memory and vary *BI*'s CXL-interleaving percentage. The results are shown in Figure 5.4, and we make two key observations. First, as more bandwidth is generated on CXL, instead of a monotonic change, the performance of the LS application initially decreases and then increases. Second, the interference becomes the worst when all memory bandwidth is generated from CXL, even though the total memory bandwidth is the lowest at this moment.

To explain the behavior in the last experiment, we need to understand how memory requests are handled in a tiered memory system. Figure 5.5 presents a high-level diagram. There exist separate fixed-size queues in the hardware for memory requests to/from local and CXL memory. Initially, when bandwidth on local memory is high, its corresponding queue fills up, causing interference on local memory (i.e., intra-tier interference) to dominate. When *BI* starts to move requests from local

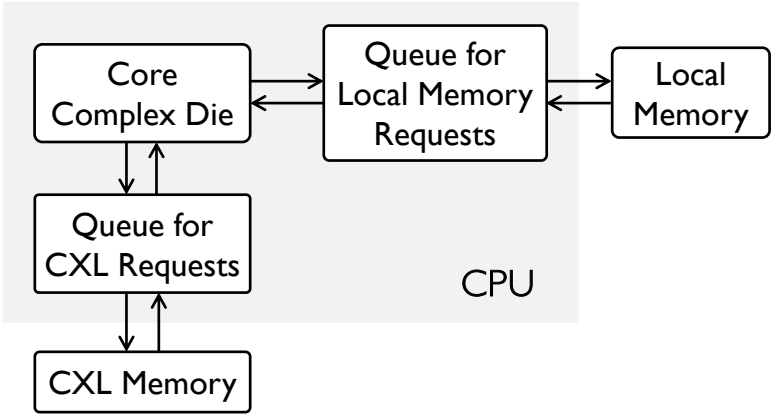


Figure 5.5: Architectural diagram of how memory requests are handled in CXL-based tiered memory.

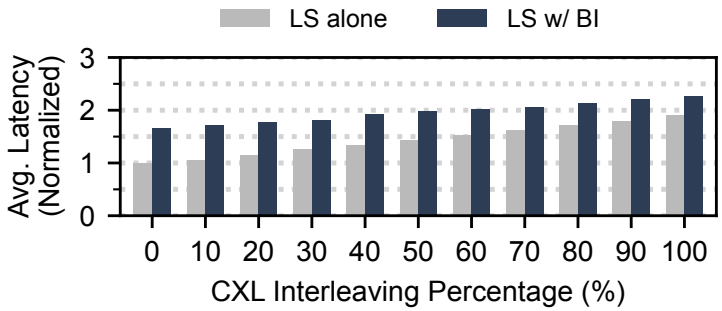


Figure 5.6: Performance of *LS* at different CXL interleaving ratios when *BI* is fixed on local memory. Migrating more requests away from local memory does not improve performance as more requests are accessing the slower tier.

memory to CXL, intra-tier interference decreases, causing latency to drop. When more requests are moved to the CXL side, the queue holding CXL requests builds up. Since both types of requests are handled by the same set of CPU cores, busy processing of the CXL requests can slow down the concurrent requests for local memory.

Takeaway #2: We should determine the right amount of memory to migrate BI applications that *reduces intra-tier interference while keeping inter-tier interference low.*

Varying *LS* across tiers. Figure 5.6 shows the results of interference in a different setting, where we keep *BI* fixed on local memory and vary *LS*'s CXL interleaving percentage. This is used to simulate the scenario where one attempts to mitigate the interference on the fast tier by migrating *LS*'s requests away from local memory to CXL memory. However, such an approach leads to worse performance because more memory requests are now accessing the slow tier.

Takeaway #3: We should proactively mitigate the interference on local memory to provide predictable performance for LS applications.

5.4 Mercury Overview

With multiple applications sharing a tiered memory, our goal in this work is to provide predictable performance among applications with different SLOs under memory resource contention and memory interference. We first discuss our design principles, followed by a system overview of Mercury.

5.4.1 Design Principles

QoS granularity. The first question we ask is *at what granularity should we support QoS and manage memory resources?* There exist multiple levels where QoS can be enforced, such as (1) an entire application, (2) individual data structures, (3) individual `mmap()` calls, or even (4) individual memory accesses. Although finer granularity allows more precise QoS assignments across different segments of the application, we build Mercury on *the application level* due to (1) easy deployment without modifying applications, (2) a clean QoS interface, and (3) low overhead in managing memory resources and performance monitoring. Moreover, providing QoS on the application level also allows us to reuse existing kernel mechanisms for identifying and migrating hot and cold pages, which has proved to be efficient [151].

Choice of performance indicators. Selecting the right performance indicators is critical for providing QoS, as Mercury needs to react quickly to meet applications' requirements. In Mercury, we prefer *low-level metrics* as performance indicators compared to application-level performance metrics. Low-level metrics can be collected via hardware-based PMU counters. Their measurements require no application modification, and as we will soon show, they closely reflect the application performance and can react faster to real-time workload changes (§5.6). In particular, we collect *memory access latency and memory bandwidth* per application. Memory access latency is measured using memory load events from L3 cache misses provided by AMD processor's Instruction Based Sampling (IBS) [70] mechanism. On the AMD platform, bandwidth is measured by μProf [26]. For intel platforms, Mercury can be applied through processor event-based sampling (PEBS) [14] where bandwidth can be measured via Intel Platform QoS (PQoS) [5].

Memory resource allocation. Instead of overprovisioning resources to handle worst-case overload scenarios, we decide to dynamically allocate *the right amount of resource that can satisfy an application's SLO*. This allows Mercury to accommodate more applications while meeting their

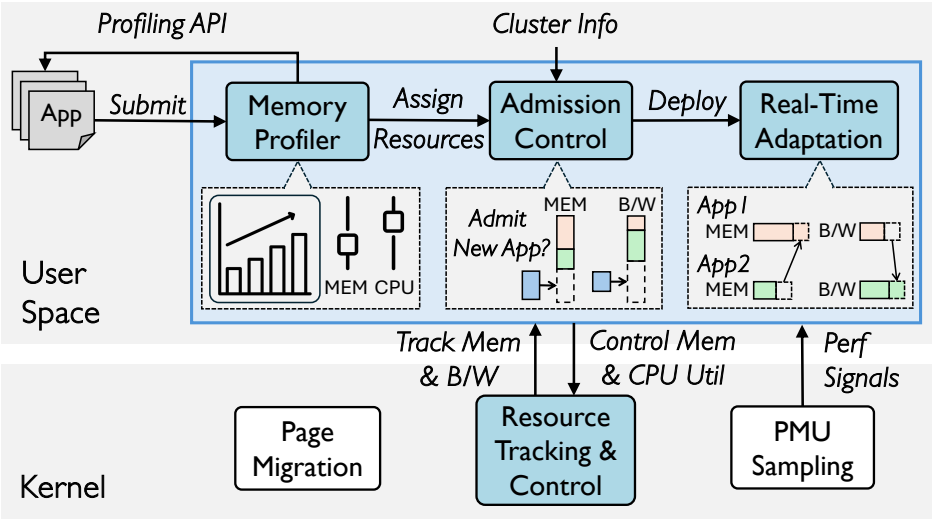


Figure 5.7: High-level system overview of Mercury. The memory profiler and the admission control determine the right resource to allocate for applications. Real-time adaptation dynamically adjusts resource allocation during runtime. The resource controller tracks and controls resources at the application level.

SLO. Following this design principle, Mercury provides QoS via a combination of *admission control and real-time adaptation*. The former admits applications using the right amount of resources, and the latter adjusts resource allocation for applications when runtime workload changes affect QoS.

Prioritization. When multiple memory-intensive applications coexist, there is no guarantee that we will always have enough resources to satisfy everyone's SLO. For example, an important application may arrive later than less critical ones, and/or an application may require more resources to fulfill increasing load. To this end, Mercury leverages *strict priority* to ensure high-priority applications get guaranteed performance even when the tiered memory runs out of resources.[1] We choose to apply a separate priority scheme on top of SLOs because more stringent SLOs do not always mean higher priority, and BI applications are not necessarily less important than LS ones. In Mercury, priority levels are uniquely assigned to avoid ties among applications.

5.4.2 System Overview

Figure 5.7 presents a system diagram of Mercury. Its design includes a profiler, an admission control algorithm, and real-time adaptation in the user space, as well as a memory resource controller implemented inside the Linux kernel.

A Mercury user (e.g., a cluster operator) submits an application along with a list of information,

[1]Lower priority applications that yield (part of) their resources to higher priority ones continue to run as best-effort to preserve work conservation.

including the number of CPU cores, memory requirement, application type, its priority level, and the SLO. The SLO of LS applications specifies a **memory access latency** target, and that of BI ones is a **maximum memory bandwidth** the application needs.

Mercury proactively controls two types of resources, **available local memory** and **cpu utilization** on the application level. Since the existing kernel mechanism on memory usage tracking and controlling does not distinguish among different tiers, Mercury enables per-tier page reclamation to control an application's available local memory (§5.5.1). Mercury adopts existing Linux page LRUs and NUMA Balancing [2] for page temperature detection and migration.

When Mercury receives the application, it first profiles offline to find the minimum amount of memory resource needed to satisfy the SLO when running in isolation (§5.5.2). The profiler also provides users with an API to measure the two SLO metrics (i.e., memory access latency and memory bandwidth) to help users set an appropriate SLO. After profiling, Mercury's admission control determines whether the new application should be admitted onto the node, and if so, how much resource we should allocate to the application, along with any changes to the existing resource allocation of other applications based on the application's priority level (§5.5.3). Mercury keeps monitoring an application's performance during its lifetime and performs real-time adaptation on its resource allocation to ensure SLO is maintained in case of workload change and bandwidth interference (§5.5.4).

5.5 Mercury Design

This section describes the details of Mercury design. We start with its application-level resource management, followed by how Mercury provides QoS with its memory profiling, admission control algorithm, and real-time adaptation.

5.5.1 Application-Level Resource Management

Mercury manages two resources – local memory and CPU utilization – to enforce SLO. The available local memory for an application determines the ratio of memory accesses on local v.s. CXL memory, which directly affects an LS application's performance. As local memory is limited and shared among all applications, Mercury decides for each LS application a *local memory limit* based on their SLO. Limiting an application's local memory can also adjust its bandwidth, as accessing more memory from a slower tier decreases the aggregate bandwidth. Meanwhile, it saves additional local memory to accommodate other applications. However, this approach alone is not enough to control an application's bandwidth for two reasons. First, placing too many memory requests on CXL memory creates inter-tier memory interference (§5.3.2). Second, the bandwidth

cannot be tuned further down after migrating all pages to CXL memory (Figure 5.3b). Therefore, Mercury also limits the CPU utilization of BI applications in addition to local memory limit in order to achieve a finer-grained control over bandwidth.

After deciding what resources Mercury manages for applications, the next step is to seek an efficient way of tracking and controlling those resources. The existing Linux kernel provides cgroup [4] to track and limit the memory and CPU usage of a collection of processes. However, the memory control mechanism in cgroup cannot be directly applied in Mercury, as it does not distinguish among different memory tiers when tracking and controlling memory usage. Instead, one can only specify a total memory limit that accounts for the total memory usage across all tiers for a given process.

Modifications to Linux cgroup. We make a series of modifications on cgroup to enable per-tier memory tracking and control. While observing the total memory usage of an application, we break it down into individual tier-level usage. In a CXL-enabled tiered memory system, memory nodes appear to the system as separate NUMA nodes. We can thus assume that each tier consists of one or multiple NUMA nodes. In this regard, we enhance the existing cgroup to track the list of pages within each LRU associated with the NUMA nodes across a specific memory tier. We, therefore, introduce a new memory limit-controlling interface, `memory.per_numa_high`, that controls the max memory usage for an application on each NUMA node. Note that `memory.per_numa_high` works concurrently with the global cgroup interface, `memory.high`, whereas the latter controls the total system-wide memory usage of an application. In contrast, our interface restricts the contribution of an application's memory footprint on each memory tier.

When allocating pages, Mercury uses Linux's default "allocate on fast memory tier first" page allocation policy unless specified otherwise. However, if the memory usage of a NUMA node within a specific memory tier exceeds its `memory.per_numa_high` threshold, Mercury stops memory allocation and initiates page reclamation only on that NUMA node. Here, the default reclamation mechanism is to demote (i.e., page migration) to the next available memory tier. Moreover, a change to `memory.per_numa_high` (e.g., by Mercury or system admins) immediately triggers reclamation across all the nodes where the new limit is below the current memory usage. Similar to TPP [151], we leverage NUMA Balancing [2] to allow page promotion among different nodes.

Mercury controls CPU utilization of an application using native cgroup's `cpu.max` interface, whose value adjusts the utilization among all CPU cores the application uses. Details of how Mercury selects an application's local memory limit and CPU utilization to provide QoS will be described next.

5.5.2 Memory Profiler

Before deploying an application, we first need to determine the right amount of memory resource needed to meet its SLO to provide QoS for more applications. In Mercury, this task is handled by the memory profiler.

Taking a user application and its SLO as inputs, the profiler finds the right amount of local memory the application needs to satisfy its SLO when running in isolation. For both types of applications, it starts by putting all their memory on the fast tier with full CPU utilization. If the SLO is not met even at this initial stage, the application is marked as *inadmissible*; the user should adjust the SLO or deploy it on another machine with larger memory or bandwidth. Otherwise, the profiler decreases its local memory limit until the measured performance matches the SLO. For BI applications, if the actual bandwidth is still above the SLO after the local memory limit drops to zero, the profiler starts to decrease CPU utilization until the SLO is met.

Note that the local memory limit found during profiling represents the minimum memory the application needs in isolation and is used for admission control (§5.5.3) to determine the local memory and CPU utilization to allocate during deployment. Mercury also records the profiled memory bandwidth for BI applications, which represents the target bandwidth to meet its initial load during admission.

Mercury also performs a one-time profiling per machine to characterize memory bandwidth interference, which will later be used by our admission control and real-time adaptation. Specifically, Mercury determines **two thresholds**, (i) a threshold on *healthy local bandwidth* ($Thresh_{local\_bw}$) and (ii) a threshold on *the rate of remote NUMA hint fault* ($Thresh_{numa}$), to monitor whether the system has reached the boundary of triggering severe intra-tier and inter-tier interference. The profiler leverages the *LS* and *BI* microbenchmarks (§**??**) to perform this task. To determine $Thresh_{local\_bw}$, the profiler launches both *LS* and *BI* on the fast tier, and increases *BI*'s bandwidth until a noticeable interference on *LS*'s latency is found. Similarly, $Thresh_{numa}$ is determined by sweeping the CXL interleaving percentage of *BI* until an obvious performance degradation is observed on *LS* running on the fast tier (10% performance degradation is set for both thresholds in our implementation).

5.5.3 Admission Control

Admission control ensures that admitted applications can coexist well in a shared tiered memory system. A naïve solution is to take each incoming application's profiling result and keep deploying applications until local memory or bandwidth capacity runs out (i.e., *first-come-first-service (FCFS)*). However, this has two drawbacks. First, a critical application may arrive late, only to find the resources have been taken by other less critical applications. Second, it does not consider the impact of bandwidth interference, which can lead to SLO violation. In Mercury, we design a new

admission control algorithm tailored for tiered memory. It leverages *strict priority scheme* to prioritize important applications, and *maximizes local memory utilization* to admit more applications while *avoiding intra-tier and inter-tier interference*.

Algorithm 4 describes Mercury's admission control. At the arrival of a new application, Mercury first checks if it is labeled as inadmissible by the profiler. If that is the case, Mercury immediately drops this application since its SLO (either latency or bandwidth) can never be met (lines 3-4). Otherwise, Mercury starts to assign resources based on the application type, as we describe below.

Admitting LS applications. Mercury directly admits an LS application if there is available local memory to satisfy its profiled local memory requirement. Otherwise, Mercury tries to find another existing application with a lower priority to *yield local memory*, starting from the one with the lowest priority (lines 7-8). However, yielding another application's memory means moving more of its requests to CXL memory, which has a risk of causing *inter-tier interference* when the victim is bandwidth-intensive (§5.3.2). Therefore, Mercury monitors the rate of NUMA hint faults and compares it with $Thresh_{numa}$ obtained from profiling (§5.5.2). If the threshold is met, Mercury stops decreasing the local memory limit of the victim application, as migrating more of its memory to CXL memory will cause inter-iter interference that harms latency performance. Instead, Mercury will reduce CPU utilization for BI apps with lower priority than the incoming app, in ascending order, until we are below $Thresh_{numa}$ (lines 9-10).

Mercury repeats this process until enough memory is yielded. If the victim application's local memory limit drops to zero, Mercury moves on to the next victim until no existing applications with lower priority can yield local memory.

Admitting BI applications. Given a BI application, the goal is to assign the right amount of local memory and CPU utilization to satisfy its bandwidth SLO. Mercury starts by assigning local memory to BI applications in the same way as it does to LS ones (lines 6-12). However, assigning the profiled local memory to the incoming application does not always mean it can achieve its bandwidth SLO, as existing applications may have already taken a large portion of the total bandwidth capacity. Under this scenario, Mercury finds existing BI applications with lower priority to *yield bandwidth*. This is done by decreasing the local memory limit of the victim application(s). During this process, Mercury checks the current rate of remote NUMA hint faults and avoids inter-tier interference by switching to decreasing the victim application's CPU utilization once $thresh_{numa}$ is exceeded (lines 16, 26-27). In the meantime, Mercury watches out for potential *intra-tier interference* when bandwidth on the fast-tier becomes too high. This is verified by checking two conditions: (1) there exist LS applications with higher priority, and (2) the healthy bandwidth threshold ($thresh_{local\_bw}$, §5.5.2) for the fast tier is exceeded. If both are met, Mercury stops assigning bandwidth on the fast tier to the incoming application (line 15, details omitted in Algorithm 4 for brevity).

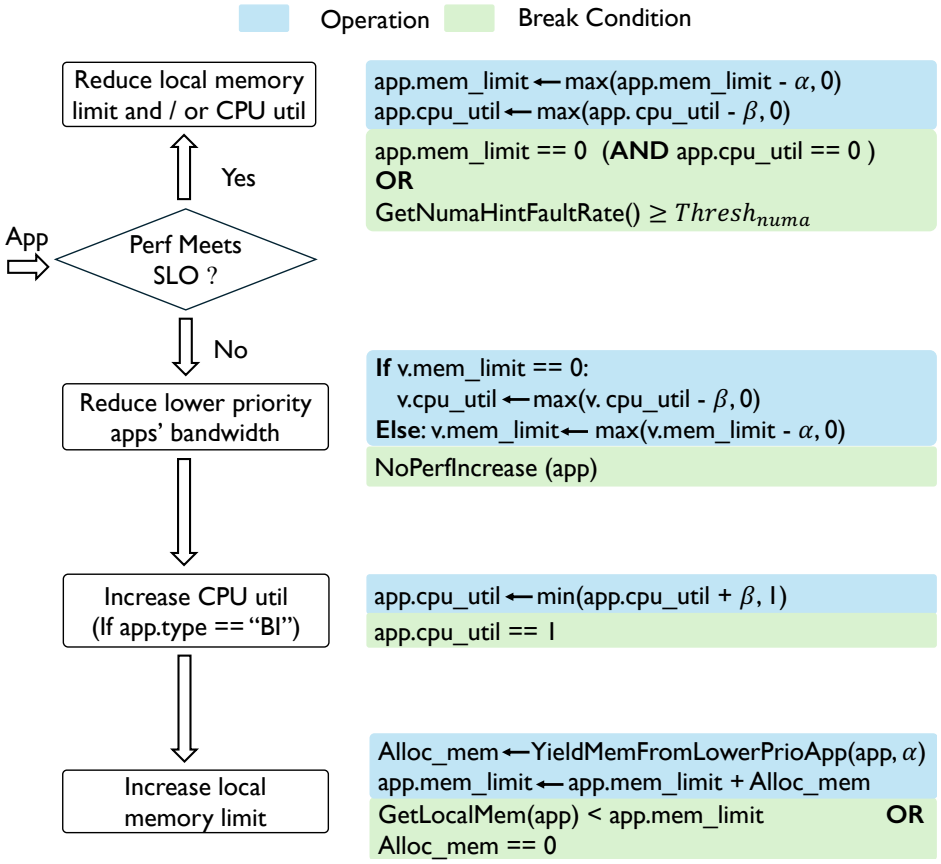


Figure 5.8: Procedure of real-time adaptation.

5.5.4 Real-time Adaptation

Although our memory profiling and admission control provide a good estimation of how many resources to assign to applications during launch time, we may still experience SLO violations in two scenarios. First, the amount of resource applications need to maintain their SLO can change over time due to workload changes. Second, LS applications may still be affected by minor interference when BI applications' bandwidth sits around one of the two interference thresholds. Mercury should adjust resource allocations quickly to ensure QoS guarantees for critical applications in such scenarios.

Figure 5.8 describes the flow of Mercury's real-time adaptation. Mercury periodically checks the performance signals (i.e., memory access latency for LS applications, and memory bandwidth for BI applications) every 200ms and verifies if they meet SLO. To ensure critical applications receive guaranteed performance, we check applications in the order of higher priority levels. If the measured performance of the target application is better than SLO, Mercury retrieves excessive resources by decreasing its local memory limit until the performance is back to expectation again. Similar to our admission control, Mercury watches out for inter-tier interference during this process

(by checking $thresh_{numa}$) to prevent too many requests from accessing CXL memory. In the case of a BI application, Mercury starts to decrease its CPU utilization when its local memory limit hits zero.

It is more challenging when the measured performance is worse than SLO, as there may be multiple possible causes. Mercury takes a series of actions to identify the right cause and make adjustments. Mercury first decreases the bandwidth of low-priority applications until no performance improvement can be detected. This step verifies if the performance drop is caused by bandwidth interference. Since all applications go through our real-time adjustment in the order of priority, Mercury ensures the BI application that generates excessive bandwidth will eventually be punished, and other BI applications will retain SLO if there is enough bandwidth. After the bandwidth check, if the measured performance is still worse than SLO, we are certain the target application needs more resources due to a workload change. In this case, Mercury allocates resources based on the type of the target application. Given a BI application, Mercury first increases its CPU utilization before increasing its local memory limit to ensure local memory is maximized among all applications. For an LS application, Mercury directly increases its local memory limit. Similar to the way in admission control, Mercury finds the local memory from low-priority applications in ascending order of the priority level.

5.6 Evaluation

We evaluate Mercury's capability of providing QoS among applications sharing a tiered memory system, in the presence of local memory contention, bandwidth interference, and dynamic workload changes. Our key findings are as follows:

(1) Mercury closely tracks SLOs for both LS and BI applications by allocating right amount of resources (§5.6.2).

(2) Mercury effectively handles local memory contention and bandwidth interference, achieving up to a 53.4% improvement compared to TPP (§5.6.3- §5.6.5).

(3) Mercury's real-time adaptation mechanism accurately reallocates resources to prioritize critical applications during runtime workload changes, resulting in 8.4× longer SLO satisfaction time compared to TPP (§5.6.6).

5.6.1 Experiment Setup

We implement Mercury's resource management module on Linux kernel v6.6. Mercury's user space components are written in C++ with 3000 lines of code.

We illustrate Mercury's effectiveness in providing QoS using four real-world applications (Table 5.1):

- *Redis (LS)*: a widely used in-memory database. We launch memtier_benchmark [3] to generate a realistic workload.
- *vectorDB (LS)*: a vector database implemented using the Faiss library [68] with FlatL2 [123] and HNSWFlat [148] indices. Each client request contains a query of 10 nearest neighbors of randomly generated vectors.
- *DLRM (BI)*: a popular deep learning recommendation model, running on Criteo Kaggle Display Advertising Challenge dataset [105]. Compared to the setting in §5.2.1 which targets low serving latency, we use another practical throughput-intensive setup where providers use a larger number of embeddings, causing high bandwidth.
- *llama.cpp (BI)*: a C/C++ implementation of inference for large language models including LLaMA [213]. We run llama.cpp on local and CXL memory with the Llama 2 70B - GGUF model [155].

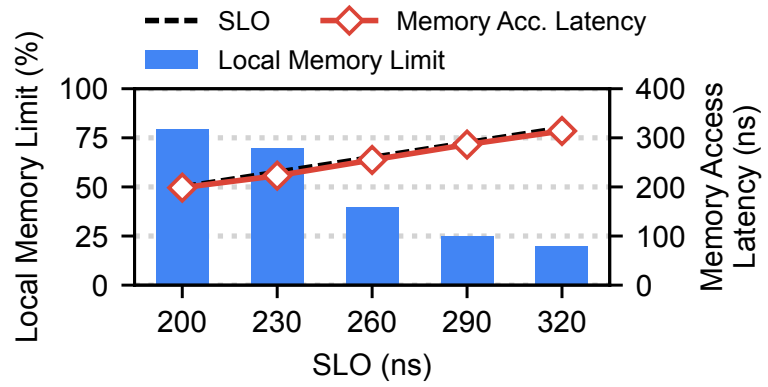
Emulation environment. We run experiments on dual-socket Intel Xeon Gold 6330 servers with 56 physical CPU cores. Each socket contains 512GB DDR4 memory. We emulate CXL by disabling all cores in one socket while keeping its memory accessible from the other socket, a method widely adopted in current CXL research [133, 151]. In the meantime, we reduce the uncore frequency on the server to match the memory access latency from the isolated socket to the latency of accessing the same CXL setup in §5.2 and §5.3.

We compare all the experiments with TPP [151], a state-of-the-art open-source tiered memory system. Each experiment is run five times, and the average measurements are reported.

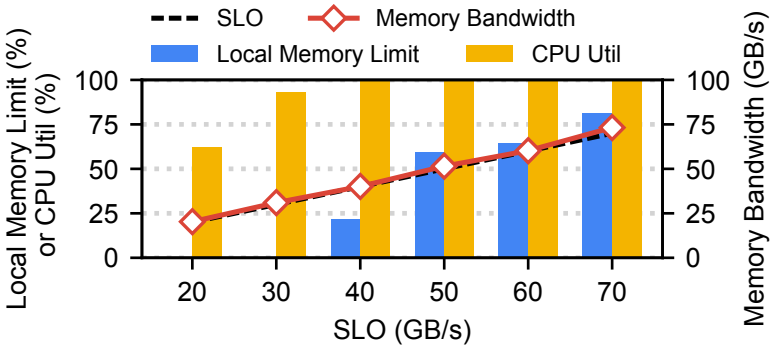
5.6.2 SLO Compliance

We start by evaluating how closely Mercury tracks SLOs of both types of applications. We first look at simple scenarios where a single application is running and leave more complicated multi-tenant settings in the follow-up subsections. Figure 5.9a records the achieved memory access latency of Redis at different SLOs, along with the local memory limit (recorded as the percentage w.r.t its 20GB WSS) Mercury sets. Mercury is able to closely track the SLO by assigning the right amount of local memory.

Mercury is also good at tracking the SLO for BI applications. Figure 5.9b shows llama.cpp's bandwidth performance and its resource allocation by Mercury under different SLOs. When bandwidth SLO is small (e.g., 30GB/s and below in this experiment), Mercury further decreases its CPU utilization, as merely migrating all memory to CXL memory(by setting the local memory limit to zero) still achieves higher bandwidth than needed. Note that in this case, Mercury does



(a) SLO compliance for Redis



(b) SLO compliance for llama.cpp

Figure 5.9: Mercury provides SLO compliance on both memory access latency (for Redis) and bandwidth (for llama.cpp).

not limit its migration for inter-tier interference, as there is no other LS application running in the system.

5.6.3 Handling Local Memory Contention

Next, we move on to multi-tenant settings and evaluate how Mercury handles local memory contention. This experiment involves a high-priority Redis process running together with a low-priority vectorDB-FlatL2 workload configured as an LS application. We set the SLO of Redis and vectorDB to be 160ns and 200ns, respectively. The WSS of Redis and vectorDB are both set to 20GB, and we configure the local memory capacity to be 20GB as well to create local memory contention. Figure 5.10 compares the average memory access latency and application-level performance between TPP and Mercury. TPP fails to achieve the SLO for Redis, as TPP cannot prevent vectorDB from stealing most of the local memory due to higher memory access frequency than Redis, leading to priority inversion. In comparison, Mercury manages to achieve

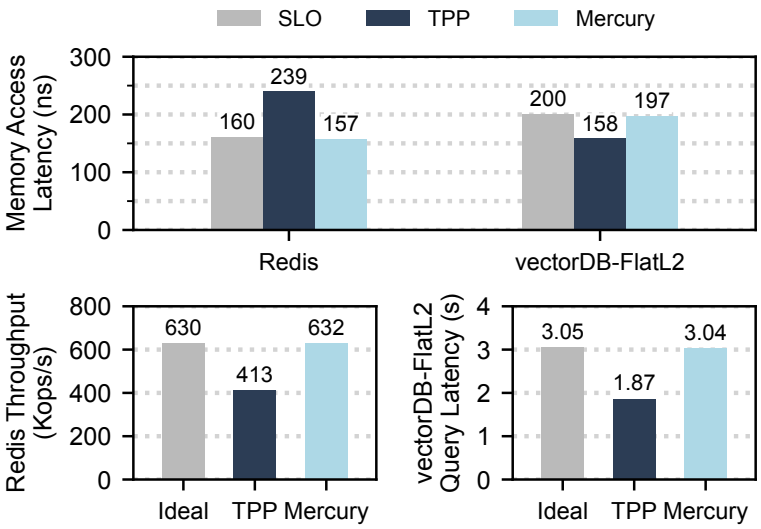


Figure 5.10: Comparing Mercury with TPP when handling local memory contention between Redis and vectorDB.

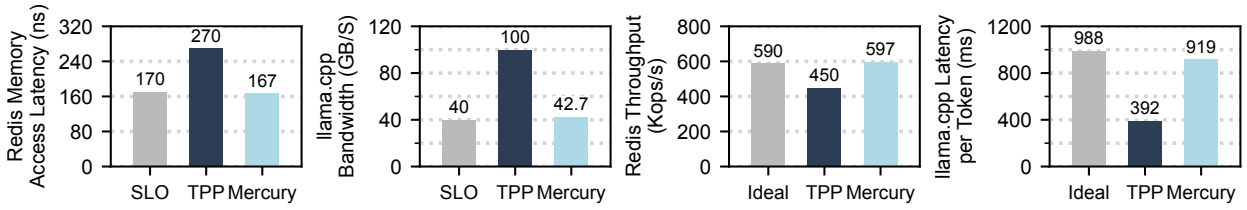


Figure 5.11: Comparing Mercury with TPP when handling memory bandwidth interference between Redis and llama.cpp.

the SLO for both applications by assigning the right amount of local memory accordingly (18GB for Redis and 2GB for vectorDB), leading to 46% improvement in the throughput of Redis over TPP.

5.6.4 Handling Memory Bandwidth Interference

Besides local memory contention, memory bandwidth interference is another key aspect Mercury tries to resolve when designing its admission control. We evaluate Mercury's effectiveness in handling bandwidth interference among different combinations of applications. In those experiments, we configure the WSS of applications such that local memory contention never happens. We start with a high-priority Redis (20GB WSS) workload running alongside a low-priority llama.cpp inference task, and record both low-level and application-level performance of both applications in Figure 5.11. TPP has no control over Mercury's bandwidth, which causes significant interference on Redis, leading to missed SLO and worse throughput. Mercury, in contrast, mitigates bandwidth

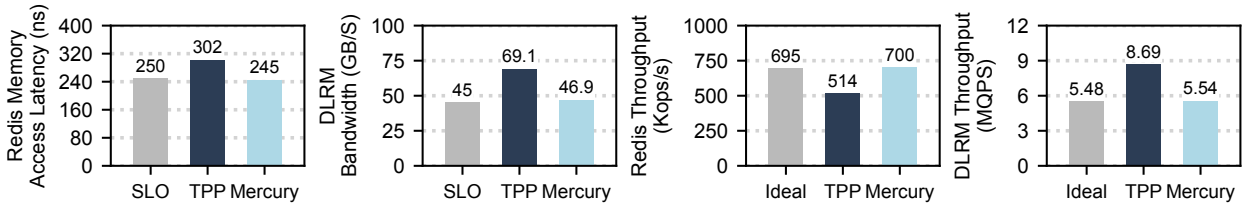


Figure 5.12: Comparing Mercury with TPP when handling memory bandwidth interference between Redis and DLRM.

interference from llama.cpp by migrating most of its memory to CXL memory (with 25.8% memory left on local), ending up with both applications' SLO satisfied and 32.7% higher throughput of Redis compared to TPP.

We observe a similar trend when launching Redis (10GB WSS) together with DLRM (Figure 5.12). Similar to llama.cpp, DLRM causes bandwidth interference that TPP cannot mitigate. Mercury adjusts DLRM's CPU utilization (64%) and local memory limit (50% of its WSS) to meet DLRM's SLO while minimizing both intra-tier and inter-tier interference, resulting in a 36.2% improvement in the throughput of Redis.

5.6.5 Mixing Two Sources of Unpredictability

It is quite likely that local memory contention and memory bandwidth interference appear simultaneously in a real-world setting. To evaluate how Mercury performs in this case, we deploy Redis, llama.cpp, and vectorDB-HNSW all together with a local memory capacity of 40GB. The WSS for Redis, llama.cpp, and vectorDB is 40GB, 40GB, and 20GB, respectively. Figure 5.13 compares Mercury with TPP on achieved low-level and application-level performance. TPP allocates almost all local memory to Redis as Redis has the highest memory access frequency among the three applications, whereas most of llama.cpp's and vectorDB's memory are placed on CXL memory. This allocation satisfies only the SLO of Redis, as llama.cpp suffers from insufficient bandwidth, and vectorDB's performance degrades due to interference from llama.cpp and not enough local memory. In contrast, Mercury satisfies the SLOs for all three applications by allocating the right amount of memory (10GB for Redis, 20GB for vectorDB, and 10GB for llama.cpp) and managing llama.cpp's bandwidth to minimize interference, resulting in a 53.4% performance improvement for vectorDB.

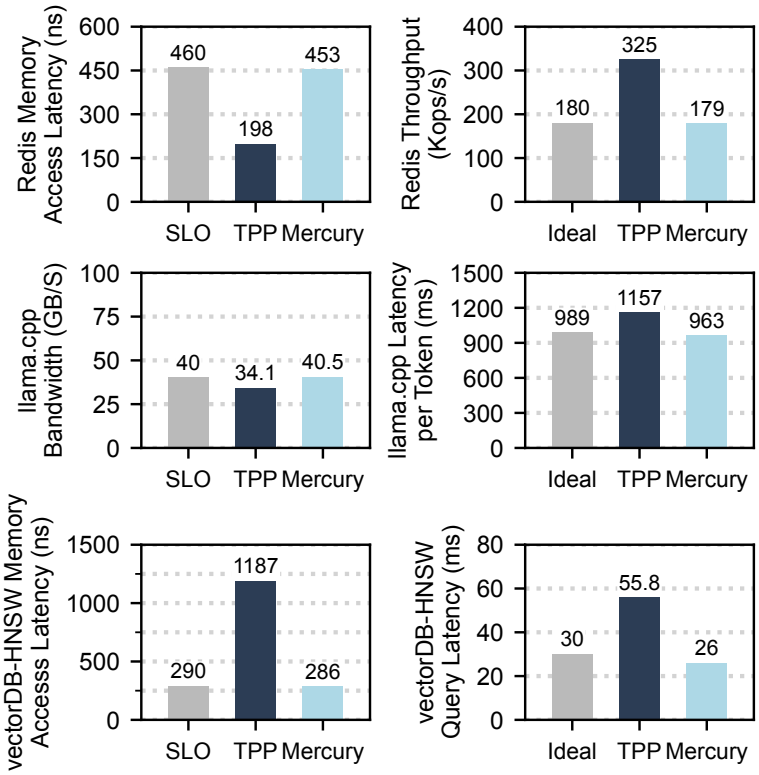


Figure 5.13: Comparing Mercury with TPP when handling both local memory contention and memory bandwidth interference among Redis, llama.cpp, and vectorDB.

5.6.6 Real-time Adaptation to Dynamic Changes

In practice, the memory or bandwidth usage of applications often changes over time. To justify Mercury's adaptability, we run Redis, llama.cpp, and vectorDB-FlatL2 together, and adjust llama.cpp's bandwidth and Redis's memory usage. The WSS for Redis, llama.cpp, and vectorDB is 30GB, 40GB, and 40GB, respectively, with the local memory capacity constrained at 70 GB. The priority level of the applications in descending order is Redis, llama.cpp, and vectorDB. We set the SLO for Redis, llama.cpp, and vectorDB to be 200ns, 70GB/s, and 180ns.

Initially, Redis and llama.cpp launch together without memory contention or bandwidth interference. When Mercury detects llama.cpp's load arrives at $t = 60s$, it quickly mitigates its bandwidth interference to ensure Redis can still maintain its SLO, as Redis has higher priority. TPP, on the other hand, gives full bandwidth capacity to llama.cpp, causing severe interference that violates the SLO of Redis. After 1100 seconds, llama.cpp's task finishes. We launch vectorDB and start to gradually increase the load of Redis. The increase in the memory usage of Redis causes local memory contention. Under TPP, Redis cannot acquire more memory due to lower access frequency compared to vectorDB. In contrast, Mercury reallocates memory from vectorDB to Redis

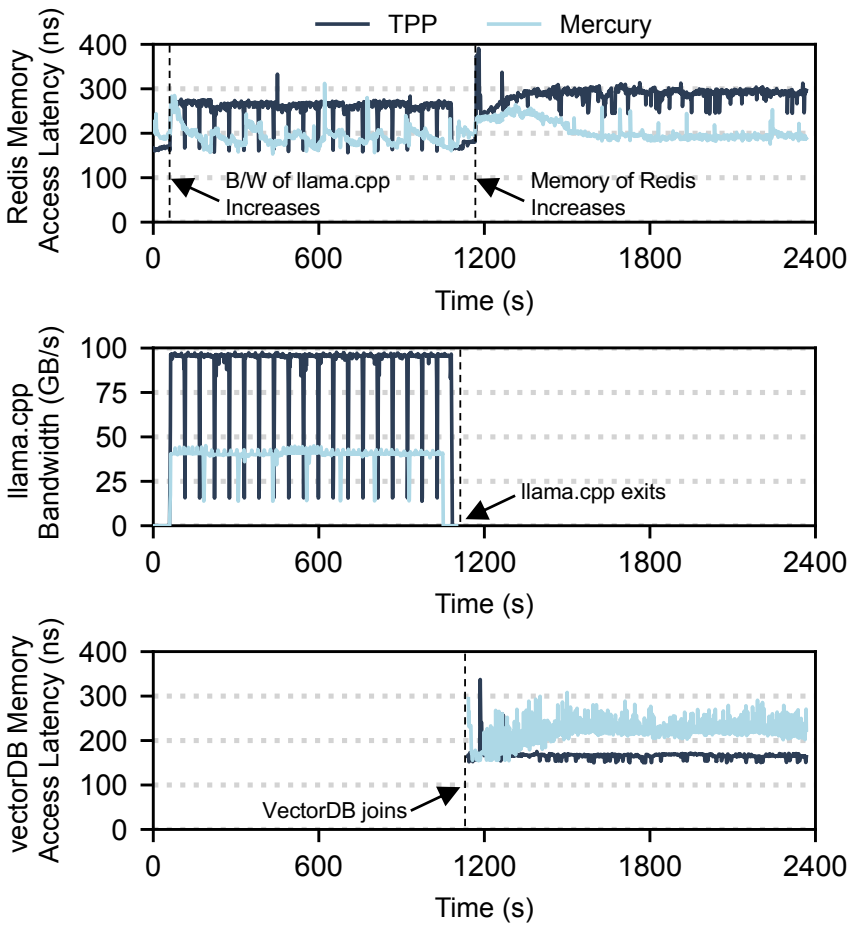


Figure 5.14: Performance of Redis, llama.cpp, and vectorDB under real-time changes. SLO for Redis/llama.cpp/vectorDB is 200ns/180ns/70GBps, with Redis having the highest priority. llama.cpp's bandwidth surges during 60-1100s; Redis's memory usage increases during 1160-2366s.

to ensure Redis meets its SLO. Throughout this experiment, Mercury results in up to 8.4× longer SLO satisfaction time for Redis compared to TPP and improved Redis's throughput performance by 33.21%.

5.7 Related Work

Tiered Memory Systems. Numerous solutions[222, 72, 184, 121] consider non-volatile memory (NVM) to be a slow memory tier. HeMem [184] introduces a flexible, per-application memory management policy at the user level. AutoTiering [121] addresses multi-tiered memory system utilization by considering access tier and locality without a predefined threshold. Diverging from NVM-focused methods, Pond [133] and TPP [151] explore CXL memory; TPP provides an OS-

level, application-transparent mechanism for CXL memory, while Pond develops a predictive model for latency and resource management in a CXL-based memory pool. Mercury decouples memory temperature from application importance, allocating local memory based on SLOs and priority and outperforming previous work that primarily considers applications with hotter pages as more important.

QoS solutions. QoS is a full-stack concern addressed across the hardware/software stack. Memshare [54] maximizes hit rates and provides isolation in multi-tenant web applications with a log-structured, application-aware design. Aequitas [229] and DiffServ [40] prioritize traffic at the network edges, with Aequitas targeting data center environments to ensure latency SLOs for RPCs. TMTS [72] uses two metrics to meet SLOs in tiered memory systems, prioritizing LS applications. Mercury introduces QoS management that ensures strict priority and fairness across application types. ZygOS [181] and Perséphone [66] leverage specialized OS design to meet tail latency SLOs for datacenter workloads. FIRM [183] leverages online telemetry data and machine-learning techniques to build a fined-grained resource management framework to provide SLO guarantees for microservices.

Interference management. Prior systems like Heracles [142] and PARTIES [48] dynamically adjust partitions to handle memory bandwidth interference, with PARTIES providing enhanced isolation for memory capacity and disk bandwidth. MCP [161] reduces inter-application interference by mapping data to separate channels, while IMPS [161] prioritizes memory non-intensive applications, which can be unfair. The FQ memory scheduler [165] prioritizes memory requests by earliest virtual finish-time. ASM [210] minimizes memory interference by periodically giving each application's requests the highest priority. However, these systems do not address inter-tier interference. Mercury is the first to combine local memory limits and CPU utilization to avoid both intra-tier and inter-tier bandwidth interference.

Disaggregated Memory. Memory disaggregation exposes capacity available in remote hosts as a shared memory pool. Recent RDMA-based disaggregated memory solutions [46, 216, 81, 90] face significantly higher latency than CXL memory [87]. Memory management in these systems is orthogonal to Mercury; one can use both CXL- and network-enabled memory tiers and apply Mercury to manage tiered memory.

5.8 Conclusion

Tiered memory systems provides higher memory capacity to allow more memory-intensive applications to be deployed. However, existing tiered memory systems focus on optimizing a single application, and cannot provide QoS guarantees when multiple applications sharing memory re-

source. We present the design and implementation of Mercury, a QoS-aware tiered memory system to provide predictable performance for coexisting memory-intensive workloads. Mercury provides application-level resource management by enabling per-tier page reclamation inside Linux kernel. By designing a novel admission control and real-time adapatation algorithm, Mercury maximizes local memory utilization while mitigate both intra-tier and inter-tier memory bandwidth interference. Mercury outperforms the state-of-the-art solution when handling local memory contention, memory interference, and dynamic workload changes with significant performance improvement.

Algorithm 4: Mercury Admission Control

Notation:
$pQueue$: queue of apps in *ascending* order of priority
$B_{slo,s}$: app s' b/w SLO, $M_{profile,s}$: app s' profiled local memory
α: memory update granularity, β: CPU update granularity

Function `DeployApp`(s):
  if !*IsAdmissible(s)* **then**
    `NotAdmit`(s)
  ▷ Assign local memory for LS & BI apps
  $v \leftarrow pQueue.top()$
  while $v \neq s$ **and** *GetLocalMemAvail()* $< M_{profile,s}$ **do**
    `YieldMem`(v, $M_{profile,s}$ − `GetLocalMemAvail()`)
    while (*GetNumaHintFaultRate()* > $Thresh_{numa}$) **do**
      `ReduceLowerPriorityAppBw`($pQueue$, s)
    $v \leftarrow v$.next()
  $s.mem\_limit \leftarrow \min($`GetLocalMemAvail()`$, M_{profile,s})$
  ▷ Assign local memory & cpu util for BI apps
  if $s.type ==$ "*BI*" **then**
    while *GetAppUsableBw(s)* $< B_{slo,s}$ **do**
      if *ReduceLowerPriorityAppBw*($pQueue$, s) **then**
        break
    $s.mem\_bw \leftarrow \min(B_{slo,s},$ `GetAppUsableBw(s)`$)$
  `LaunchApp`(s)
  $pQueue.enqueue(s)$

Function `ReduceLowerPriorityAppBw`($pQueue$, s):
  $v \leftarrow pQueue.top()$
  while $v \neq s$ **do**
    if $v.type ==$ "*LS*" **or** (!$v.cpu\_util$ **and** !$v.mem\_limit$) **then**
      $v \leftarrow v$.next()
    if !*IsInterTierHealthy*(s) **or** !$v.mem\_limit$ **then**
      $v.cpu\_util \leftarrow \max(0, v.cpu\_util - \beta)$
    else
      $v.mem\_limit \leftarrow \max(0, v.mem\_limit - \alpha)$
    return **true**
  return **false**

Table 5.1: Four real-world applications used in evaluations.

Application	Application Level Metric	Type
Redis	Operations Per Second	LS
vectorDB	Latency per Query	LS
llama.cpp	Latency Per Token	BI
DLRM	Queries Per Second	BI

CHAPTER 6

Conclusion

Existing system designs have been focusing on improving system performance and efficiency to optimize single or the same type of applications running in isolation. However, as cloud infrastructure continues to scale, multiple applications have to share cloud resource, calling for quality-of-service support to ensure critical applications receive desired performance. This is not an easy task, as we must also ensure high resource utilization is achieved in order not to waste cloud resources. This requires system designers to efficiently resolve resource contention and allocate resource in a fine-grained manner.

Over the past few years, we have identified several areas in cloud infrastructure that lack QoS support, and have built QoS support in network interface cards, datacenter fabrics, edge devices, and tiered memory systems. We show that it is possible to achieve predictable performance and high utilization at the same time. Justitia (Chapter 2) provides performance isolation in KBNs for latency- and throughput-sensitive applications, while still maximizing the link utilization for bandwidth-intensive applications. Aequitas (Chapter 3 provides RPC network latency guarantees among different QoS levels for critical RPCs in datacenter fabrics, and maximize the amount of RPC traffic admitted into each QoS level. Vulcan (Chapter 4) generates optimal query plan for live ML queries that is able to not only satisfy their accuracy and latency SLOs but also minimize their resource consumption across edge tiers. Mercury (Chapter 5) is a QoS-aware tiered memory systems that provides SLO to critical applications while maximizing local memory utilization on the fast tier, such that more memory-intensive applications can be admitted.

In the rest of this chapter, we summarize the common design principles we learned when designing the four QoS-aware systems in this dissertation, followed by a discussion of future work.

6.1 Common Design Principles for Building QoS-aware Systems in the Cloud

Identifying source of performance unpredictability All of the QoS research in this dissertation starts from identifying the source of performance unpredictability. This allows us to locate possible resource contention in the system, along with other factors that affect QoS. In Justitia, we perform a detailed analysis on performance anomalies in KNB to identify multi-resource contention inside the NIC, along with the head-of-line blocking that hinders latency-sensitive applications. This leads to design of the multi-resource token and message splitting in Justitia. In Aequitas, we perform a thorough theoretical analysis on QoS distribution v.s. worst-case delay. The key design idea of Aequitas' distributed admission control is inspired by our theoretical results. Similarly, we follow the same principle in Mercury and identify local memory contention and inter-tier bandwidth interference, based on which we design its admission control and real-time adjustment algorithm. Vulcan also benefits from this principle by discovering the impact of filtering ordering on query performance, a key observation we leverage in constructing the ML pipeline.

Selecting the right performance indicator. Selecting the right performance indicators is critical for providing QoS. QoS-aware systems need to monitor application performance to ensure their SLOs are maintained, and react quickly when resource contention happens. A good performance indicator should be easy and fast to collect, and can reflect the change of the hard-to-collect application-level performance. Justitia leverages the reference flow to monitor latency-sensitive applications as they themselves send sparse messages, which is too slow to detect performance chnage. Aequtas select RPC network latency to focus on the network component of the RPC completion time, in order to filter out other components that are not affected by network overloads. Vulcan defines a utility function to combine query accuracy, end-to-end latency, and resource consumption to perform multi-target optimization. Mercury selects low-level memory access latency and bandwidth collected from hardware-based performance monitoring counters as they are fast to collect and closely reflect application-level metrics.

Leveraging fined-grained control on resource. Fined-grained control over resource is the key to predictable performance. An ideal resource controller should apply direct control over critical resource in the system, and allocation should be easy and quick. Justitia contorls NIC resources via its multi-resource token, which works on the operation level and thus efficient enough to control ultra fast RDMA messages. Aequitas directly controls QoS distribution via its admission control, which directly affect the delay bound for RPCs among different QoS levels. Vulcan performs a combinations of analysis to come up with the optimal query configuration that allocate each pipeline component's resource based on its pipeline construction, placement, and configuration.

Mercury determines the right amount of resource to meet SLO for both latency-sensitive and bandwidth-intensive applications via its kernel-level resource controller.

6.2 Future Work

In this section, we discuss future directions of building QoS-aware systems for cloud applications.

Co-designing Justitia with congestion control. Although Justitia effectively complements DCQCN (§2.6.3) in simple scenarios, DCQCN considers only bandwidth-sensitive flows. A key future work would be a ground-up co-design of Justitia with DCQCN [237] or TIMELY [156] to handle all three traffic types for the entire fabric with sender- and receiver-side contentions (§2.6.5). While network calculus and service curves [104, 208, 60, 61] dealt with point-to-point bandwidth- and latency-sensitive flows, their straightforward usage can be limited by multi-resource RNICs and throughput-sensitive flows. At the fabric level, exploring a Fastpass-style centralized solution [177] can be another future work.

Application-based Aequitas While Aequitas provides SLOs for all admitted RPCs, it does not guarantee the amount of traffic admitted on a per-application basis. The admitted traffic depends on the number of coexisting applications, as Aequitas shares the per-QoS bandwidth. This requires modification on the admission control, as now we need to treat all RPCs from the same application as a united entity. One possible solution to provide application traffic rate guarantees is to leverage a centralized RPC quota server, where applications are only allowed to use certain QoS levels with quota. This idea also add challenges in delay bound calculation, as know we also need to consider additional rtt to fetch the quota from the centralized server.

Multi-resource sharing among concurrent ML pipelines. Vulcan currently does not support concurrent execution of different ML queries sharing the compute and network resource across the edge tiers. In particular, Vulcan might not generate optimal query plans because each pipeline is considered separately. When multiple ML pipelines shares the edge tier, the query plan given by Vulcan can allocate multiple component to the same edge tier or network, causing resource contention. This new direction requires even larger search space, if ones desired to find the optimal query plan among all coexisting queries. However, it is a very practical problem for live ML queries as ML analytics continue to scale.

APPENDIX A

A.1 Hardware Testbed Summary for Justitia

Table A.1 summarizes the hardware we use for different RDMA protocols in our experiments.

A.2 Characteristics of Latency- and Throughput-Sensitive Applications in the Absence of Bandwidth-Sensitive Ones

Multiple latency-sensitive applications can coexist without affecting each other (Figure A.1). Although latencies increase, everyone suffers equally. All applications experience the same throughputs as well.

Similarly, multiple throughput-sensitive applications receive almost equal throughputs when competing with each other, as shown in Figure A.2.

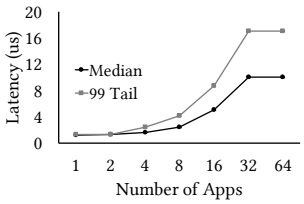
Finally, throughput-sensitive applications do not get affected by much when competing with latency-sensitive applications (Figure A.3c). Nor do latency-sensitive applications experience noticeable latency degradations in the presence of throughput-sensitive applications except for iWARP (Figure A.3a and Figure A.3b).

A.2.1 Adding More Competitors Exacerbates the Anomalies

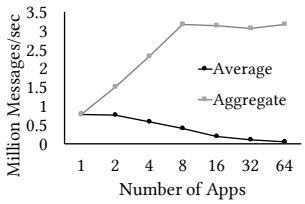
The lack of protection for the latency-sensitive applications further exacerbates as more bandwidth-sensitive applications (or equivalently more QPs) are created. We increase the number of bandwidth-sensitive applications (each with a single QP) in our experiment to simulate more realistic datacenter applications. Although InfiniBand performs relatively well in the presence of a single background bandwidth-sensitive application (Figure 2.3), adding one more competitors incurs an additional drop of $2.65\times$ and $3.79\times$ in median and 99th percentile latencies (Figure A.4a).

Protocol	NIC	Switch	NIC Capacity
InfiniBand	ConnectX-3 Pro	Mellanox SX6036G	56 Gbps
InfiniBand	ConnectX-4	Mellanox SB7770	100 Gbps
RoCEv2	ConnectX-4	Mellanox SX6018F	40 Gbps
RoCEv2 (DCQCN) (§2.6.3)	ConnectX-4 Lx	Dell S4048-ON	10 Gbps
iWARP	T62100-LP-CR	Mellanox SX6018F	40 Gbps
RoCEv2 (DCQCN) (§2.6.5, §2.6.6)	ConnectX-3 Pro	HP Moonshot-45XGc	10 Gbps

Table A.1: Testbed hardware specification.



(a) Latency



(b) Throughput

Figure A.1: Latencies and throughputs of multiple latency-sensitive applications in InfiniBand. Error bars (almost invisible due to close proximity) represent the applications with the lowest and highest values.

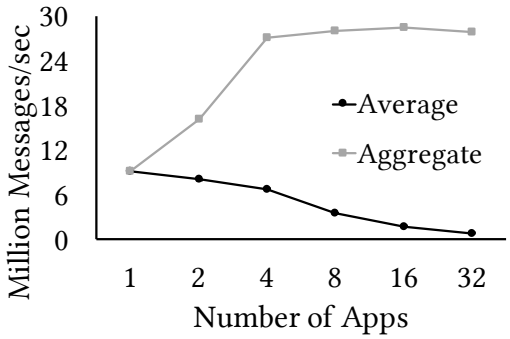
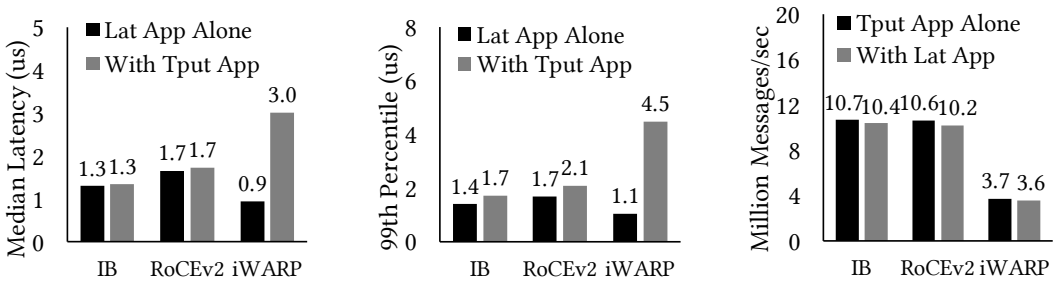


Figure A.2: Throughput of multiple throughput-sensitive applications in InfiniBand. Error bars (almost invisible due to close proximity) represent the applications with the lowest and highest values.

With 16 or more bandwidth-sensitive applications, the latency-sensitive application can barely make any progress. We observed a similar trend in other RDMA technologies.

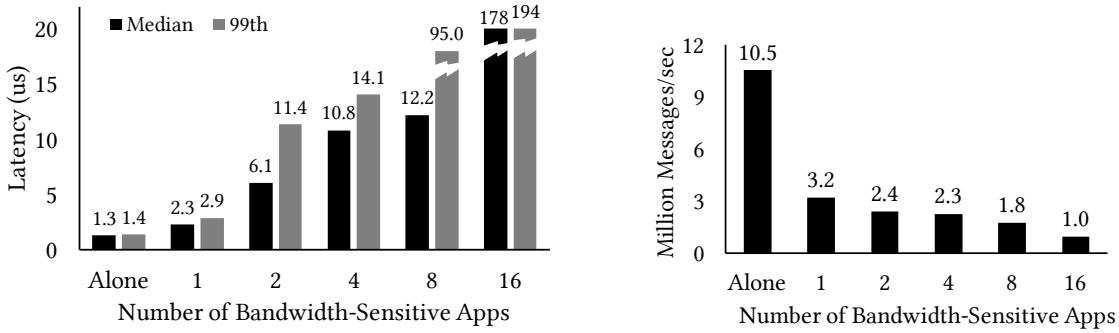
Similarly, a throughput-sensitive application loses 90% of its original throughput with 16 bandwidth-sensitive applications (Figure A.4b).

Those anomalies illustrate RNIC's inability to handle multiple types of applications, which could stem from the limited number of queues inside the RNIC hardware, increasing Head-of-Line



(a) Latency App (Med) (b) Latency App (99th) (c) Throughput App

Figure A.3: Performance anomalies of a latency-sensitive application running against a throughput-sensitive application.



(a) Impact on Latency (b) Impact on Throughput

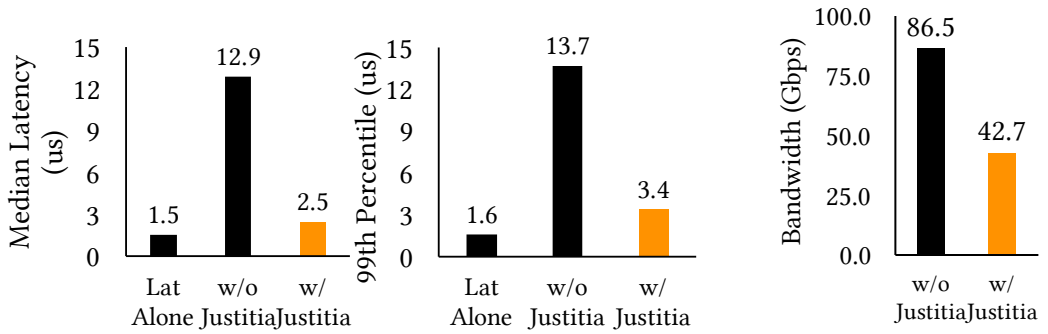
Figure A.4: Impact of increasing background bandwidth-sensitive applications (sending 1MB messages) in InfiniBand.

blocking of small messages.

A.3 Additional Evaluation Results

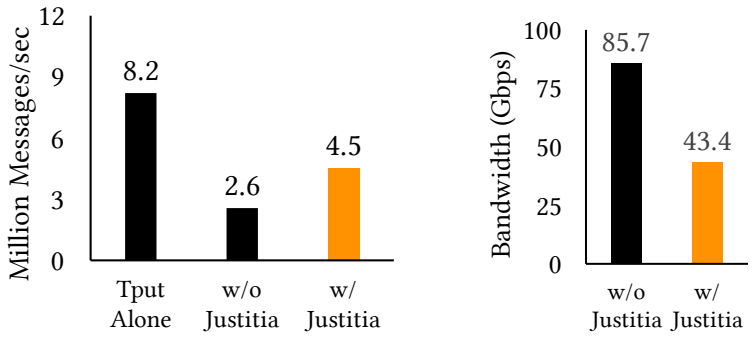
A.3.1 100 Gbps Results With/Without Justitia

Similar to the anomalies observed for 10, 40, and 56 Gbps networks (§2.3), Figure A.5 and Figure A.6 show that latency- and throughput-sensitive applications are not isolated from bandwidth-sensitive applications even in 100 Gbps networks. In these experiments, we use 5MB messages since 1MB messages are not large enough to saturate the 100 Gbps link. Justitia can effectively mitigate the challenges by enforcing performance isolation.



(a) Latency-sensitive App (b) Bandwidth App

Figure A.5: [100 Gbps InfiniBand] Performance isolation of a latency-sensitive application against a bandwidth-sensitive application.



(a) Throughput App (b) B/w App

Figure A.6: [100 Gbps InfiniBand] Performance isolation of a throughput-sensitive application against a bandwidth-sensitive application.

A.3.2 Real RDMA-based Systems Require Isolation

Besides DARE, highly-optimized RDMA-based RPC systems also suffer from unmanaged RNIC resources. Here we pick two representative systems, FaSST [112] and eRPC [113], to illustrate why they require performance isolation and how Justitia effectively achieves it. To generate background traffic, we implemented a simple RDMA-based blob storage backend across 16 machines. Users read/write data to this storage using a PUT/GET interface via frontend servers. Objects larger than 1MB are divided into 1MB splits and distributed across the backend servers. This generates a stream of 1MB transfers, and the following RDMA-optimized systems have to compete with them in our experimental setup.

FaSST is an RDMA-based RPC system optimized for high message rate. We deploy FaSST in 2 nodes with message size of 32 bytes and a batch size of 8. We use 4 threads to saturate

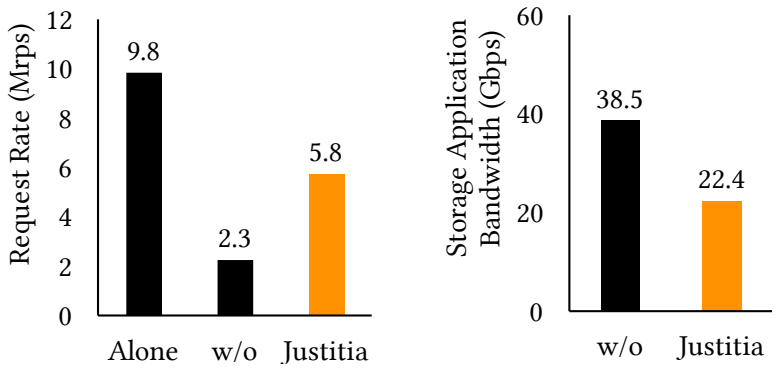


Figure A.7: Performance isolation of FaSST running against a bandwidth-sensitive storage application.

FaSST's message rate at 9.8 Mrps. In the presence of the storage application, FaSST's throughput experiences a 74% drop (Figure A.7).

eRPC is an even more recent RPC system built on top of RDMA. We deploy eRPC in 2 nodes with message size of 32 bytes. We evaluate eRPC's latency and throughput using the microbenchmark provided by its authors. For the throughput experiment, we use 2 worker threads with a batch size of 8 on each node because 2 threads are enough to saturate the message rate in our 2-node setting. In the presence of the storage application, eRPC's throughput drops by 93% (Figure A.8b), and its median and tail latencies increase by 67× and 40×, respectively (Figure A.8a).

By applying Jusitita, FaSST's throughput improves by 2.5× (Figure A.7). Justitia also improves eRPC's median (tail) latency improves by 56.9× (32.2×) and its throughput by 9.7× (Figure A.8). Note that the throughput of the storage applications drops to half of the maximum throughput in both cases because we treat the background application as a whole (and thus with equal weights to all applications, the $SafeUtil$ is $\frac{1}{2}$ of the line rate), which is different from how we treat the parallel writes in the case of Apache Crial.

A.3.3 Handling Remote READs

RDMA READ verbs can compete with WRITEs and SENDs issued from the opposite direction (§2.4.5) Figure A.9 shows that Justitia can isolate latency-sensitive remote READs from local bandwidth-sensitive WRITEs and vice versa.

A.3.4 Justitia vs. LITE

LITE [214] is a software-based RDMA implementation that adds a local indirection layer for RDMA in the Linux kernel to virtualize RDMA and enable resource sharing and performance isolation. It can use hardware virtual lanes and also includes a software-based prioritization scheme.

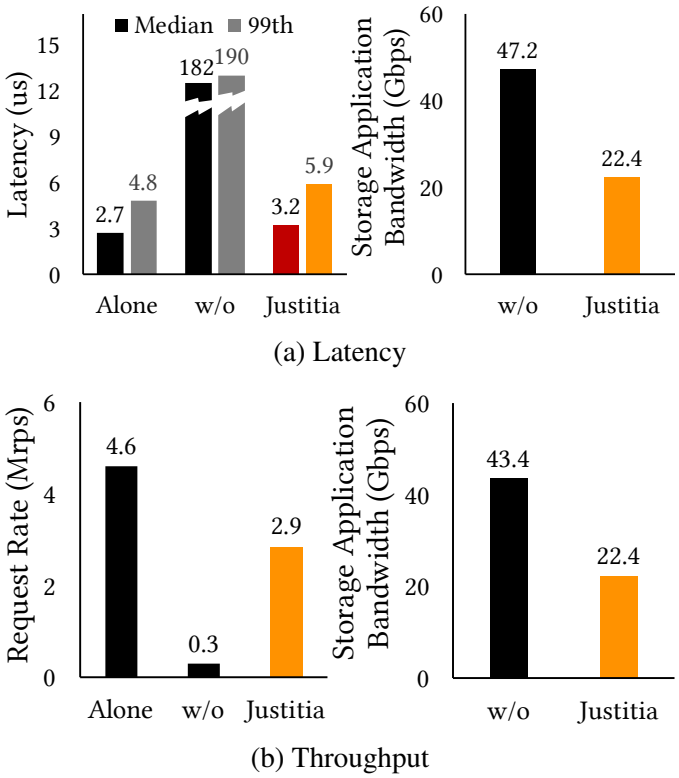


Figure A.8: Performance isolation of eRPC running against a bandwidth-sensitive storage application.

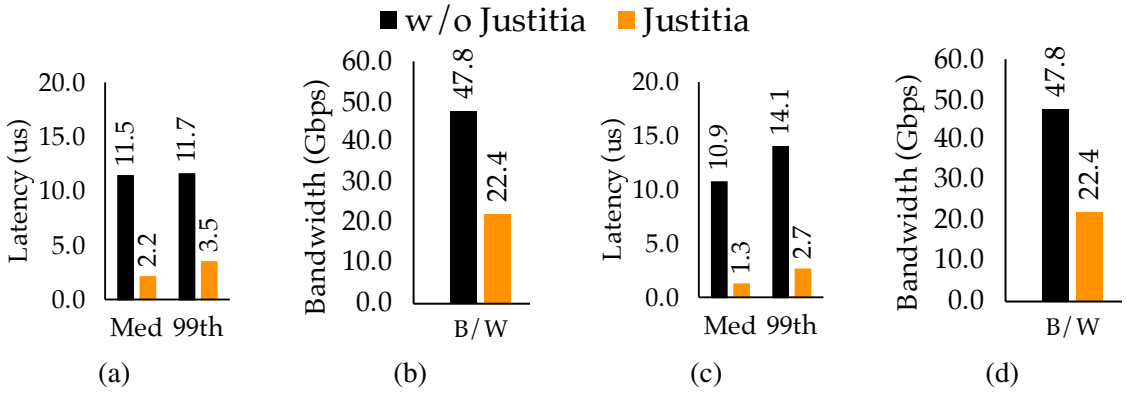
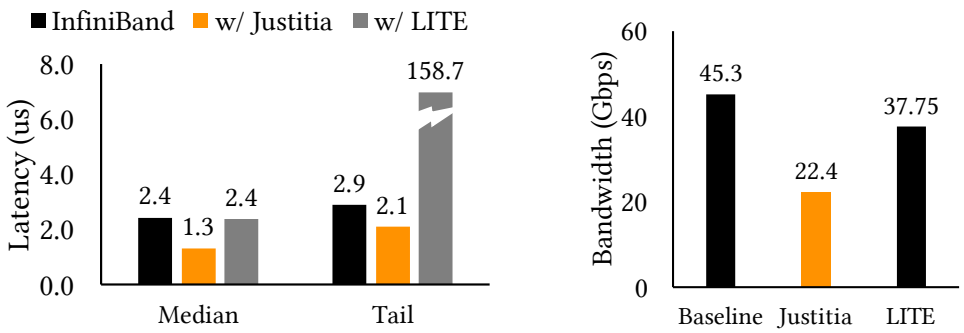


Figure A.9: [InfiniBand] a–b Justitia isolating remote latency-sensitive READs from local bandwidth-sensitive WRITEs. c–d Justitia isolating local latency-sensitive WRITEs from remote bandwidth-sensitive READs.

We found that, in the absence of hardware virtual lanes, LITE does not perform well in isolating latency-sensitive flow from the bandwidth-sensitive one (Figure A.10) – 122× worse 99th percentile latency than Justitia. In terms of bandwidth-sensitive applications using different message sizes,



(a) Latency-sensitive App (b) Bandwidth App

Figure A.10: [InfiniBand] Performance isolation of a latency-sensitive flow running against a 1MB background bandwidth-sensitive flow using Justitia and LITE.

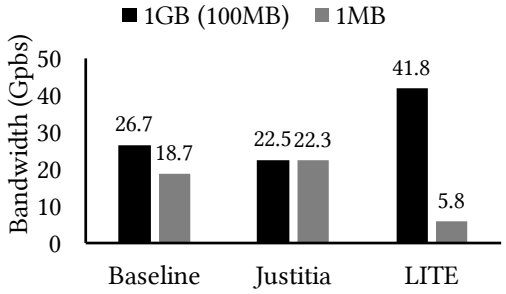
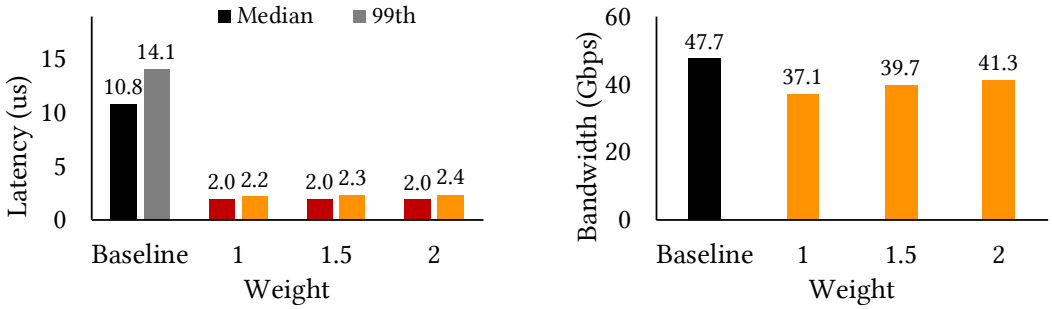


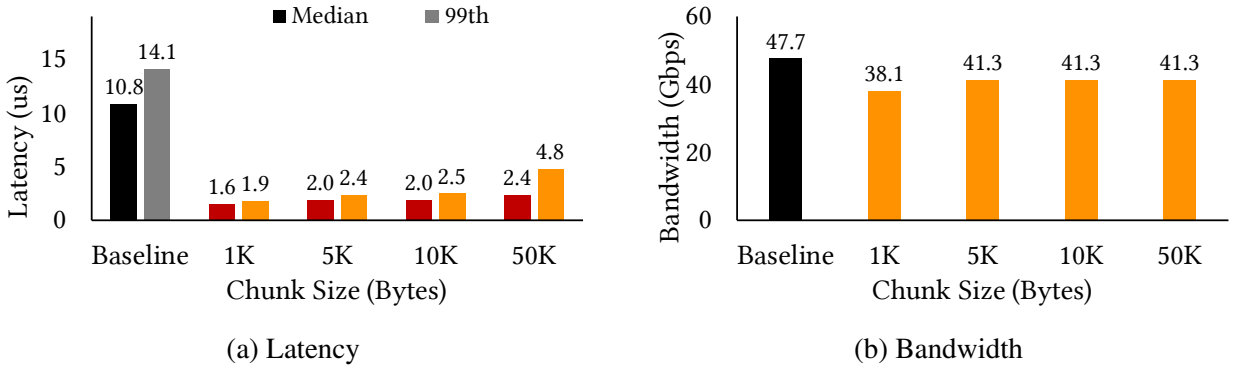
Figure A.11: [InfiniBand] Bandwidth allocations of two bandwidth-sensitive applications using Justitia and LITE. LITE uses 100MB messages instead of 1GB due to its own limitation.



(a) Latency (b) Bandwidth

Figure A.12: [InfiniBand] Sensitivity analysis of application weights.

LITE performs even worse than native InfiniBand (Figure A.11). Justitia outperforms LITE's software-level prioritization by being cognizant of the tradeoff between performance isolation and high utilization.



(a) Latency

(b) Bandwidth

Figure A.13: [InfiniBand] Sensitive analysis of chunk sizes.

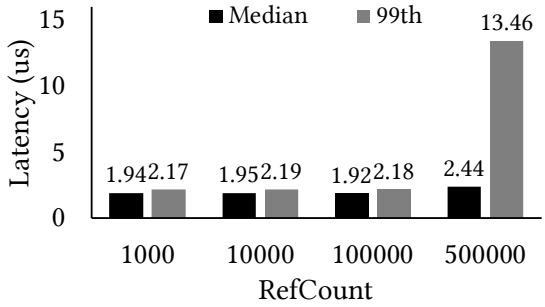


Figure A.14: [InfiniBand] Sensitivity analysis of RefCount.

A.4 Sensitivity Analysis

Setting Applications Weights To evaluate how assigning different application weights (§2.4.3.1) affects Justitia's performance, we launch 4 bandwidth-sensitive applications each sending 1MB message together with a latency-sensitive application, and we vary the weights of the bandwidth-sensitive applications. Figure A.12 illustrates the impact of setting different application weights with latency target set to 2 μs. As the weight increases, the value of $SafeUtil$ increases, and thus more aggregate bandwidth share is obtained for bandwidth-sensitive applications. Higher $SafeUtil$ leads to worse latency isolation, but in these experiments the effect of weights on tail latency performance is not huge. In fact, we do not find much latency performance degradation as the weight increases, illustrating the effectiveness of Justitia mitigating head-of-line blocking via its splitting mechanism. The cluster operator can choose weights based on the priority of the applications in the cluster based on the Quality-of-Service inside the cluster (similar to deciding bandwidth resources in a shared environment), or based on how much each application pays to obtain the service. Additionally, if multi-tenant fairness is desired, one can achieve that by modifying how credits are allocated in Justitia on a per-tenant basis. Justitia supports allocating

tokens at multiple granularities if needed, which can be per-tenant, per-application, or per group of connections within an application.

Setting Chunk Size When latency-sensitive applications are present, Justitia picks the smallest chunk size that still provides a wide range of bandwidth in case $SafeUtil$ is high (Figure 2.10). Here we evaluate how setting the correct chunk size affects Justitia's performance. We use the same setting as the sensitivity analysis of application weights and set the weight to be 2. Figure A.13 illustrates the experiment results with different chunk sizes. Although a smaller chunk size provides better latency isolation, it is not able to achieve $SafeUtil$ and thus waste bandwidth resources. On the other hand, too big of a chunk size does not provide enough latency isolation.

Setting *RefCount* To evaluate how sensitive the value $RefCount$ (§2.4.3.2) is, we design an experiment where initially we launch one latency-sensitive application to compete with a bandwidth-sensitive application. Once the experiment starts, we add three additional bandwidth-sensitive application, with a gap of 1 second between their arrival time. We measure the latency of latency-sensitive application after it completes 10 million messages. Figure A.14 plots the results with different $RefCount$ values. It turns out that Justitia tracks the tail latency closely as long as $RefCount$ is not huge. In Justitia, we set the default value of $RefCount$ =10000 to have some memory of latency spikes but not longer enough to impact stable performance.

A.5 Discussion

A.5.1 Why not simply use hardware priority queues in the RNIC?

Mellanox NICs have priority queues, but as we mention in Chapter 2, the number of queues they support is very limited (e.g., only 2 lossless queues in the RoCE NICs we test out), and we have illustrated such limited number priority queues are insufficient to provide isolation in Figure 23. In addition, the time needed to reconfigure and modify the mapping from applications' QPs to the priority queues is in the order of milliseconds. Last but not the least, it is sometimes also desireable to provide isolation inside a priority level (e.g., bandwidth-sensitive applications and latency-sensitive applications are both assigned with the same QoS level) where hardware priority queues will not be sufficient. Thus, using the priority queues provided by existing hardware does not solve the isolation problem that Justitia faces.

A.5.2 Why use only 1 QP in most of the microbenchmark experiments?

We use a small number of QPs to show that the performance isolation issues can easily occur even with a very small number of active connections. We also test with more number of QPs but the results are placed in Appendix due to limit of space. In fact, adding more QPs exacerbates the performance degradation (Figure 30 in the appendix).

A.5.3 How does Justitia handle the incast experiment?

Justitia leverages receiver-side updates to make sure the correct minimum rate guarantees are updated correctly at each sender. Due to large latency spike in the case of a network incast, senders will mostly like send via the minimum guaranteed rate (R_{min}) given the latency target will not be met. We discussed receiver-side updates in Section 2.4.5 and illustrate Justitia complements with existing congestion control and can further help reduce receiver-side engine congestion in Section 2.6.5.

A.5.4 Does reference flow and receiver-side updates create additonal congestion in a large scale deployment?

The reference flow sends small messages (10 Bytes every 20 μs) and only amount to a very small Gbps number (1e6 / 20 * 10 / 1e9 * 8 = 0.004 Gbps), which consumes less than 0.1% of the total link capacity even at nodes with only 10 Gbps link, and thus is not likely to generate any hot spot in the network. When the server broadcasts the receiver-side update, the message is sent using SEND and RECV with a message size of 16 Bytes. With even 1000 client machines this amounts to around 16KB total message size, which is too small to create a potential congestion problem.

In the case of a large-scale latency-sensitive flow incast, if congestion indeed happens, DCQCN will work together with Justitia since it is the major congestion control dealing with fabric congestion. In this scenario, adding more latency-sensitive flows does not prevent Justitia guaranteeing bandwidth share of bandwidth hungry applications.

In the current design of Justitia, the bandwidth-sensitive applications can be rate-limited due to a coexisting latency-sensitive application which is launched at the same host but sends data to a different destination. This is intended behavior to mitigate the anomalies caused by contention at sender-side RNICs, which happens regardless of whether two competing applications are targeting the same receiver. We defer a comprehensive fabric-level solution which involves multiple senders and receives as our future work.

A.5.5 How to ensure all cooperating SW uses the right protocols and protocol versions?

To deploy Justitia, one only needs to install the Justitia Daemon code and a modified Mellanox driver code on the host machine, and Justitia is compatible with all existing RDMA protocols, including RDMA over Infiniband and RoCE. In datacenter deployments, cluster management tools like Ansible can be used to ensure the appropriate code is deployed at each machine. Additionally, it is straightforward to upgrade Justitia. Because each server in Justitia operates independently, it is not necessary for the same version of Justitia to be deployed across the cluster. Justitia will operate as long as servers are running some version of Justitia.

A.5.6 How can Justitia be implemented in hardware?

Without having a software layer to split the large RDMA operations before they arrive at the NIC, one probably need to somehow control how the NIC issues PCIe reads. Hardware is often optimized for performance, which in fact is why we are having such isolation issues, so simply decreasing the size of each PCIe reads will definitely affect its maximum throughput performance. To bring Justitia into the hardware design, similar to what we have done in the software layer, the hardware need to recognize when splitting and pacing is needed to provide isolation, and when it should process at maximum capacity for higher utilization.

A.5.7 Long-term value of Justitia

As RNICs keep evolving, its performance isolation issues may be mitigated in newer hardware designs. The purpose of this work is to show that there exist such isolation issues in current kernel-bypass networks and illustrate one working approach to mitigate the issue. Design ideas presented in this work can inform hardware designers when developing future RNIC as well as programmable NIC designs.

APPENDIX B

B.1 Measuring RNL

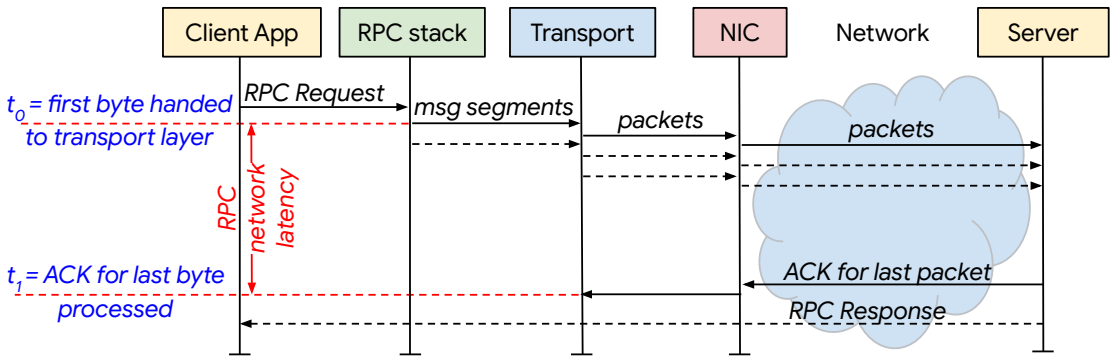


Figure B.1: Latency breakdown of a WRITE RPC. The RPC network latency (RNL) measures the difference between the time when the first RPC packet arrives at the L4 layer and the time when the last RPC packet is acknowledged.

RNL is the portion of RPC latency impacted by network overloads. Figure B.1 shows the life of a Storage WRITE RPC; the discussion applies similarly to READ RPCs. A complete Storage RPC operation consists of a request followed by a response. As Figure B.1 shows, it constitutes a significant portion of overall RPC latency, especially when network bandwidth is constrained. RNL is defined by $t_1 - t_0$ where t_0 is the time when the first RPC packet arrives at the L4 transport layer, e.g., TCP and t_1 is the time when the last packet of the RPC is acknowledged at the transport. We focus on the payload portion of the RPC as we observe that total data transmitted in a complete RPC operation (a request followed by a response) is dominated by the side which contains the actual payload of the RPC—the response of a READ RPC is much larger than the request (200:1 on average in our clusters), and the request of a WRITE RPC is much larger than its response (400:1 on average).

There are two main challenges in precisely measuring RNL in production stacks: (i) RPC boundaries may not be known precisely at the transport layer, as is the case with Linux kernel TCP, and (ii) RNL can still include delays unrelated to network overload, such as delays due to insufficient CPU, interrupts, flow control or kernel scheduler. One approach is to measure RNL t_0 and t_1 at the *sendmsg* boundary. As Aequitas incorporates RPC size in the SLO, this works well even if an RPC consists of multiple $sendmsg$ calls.

B.2 WFQ Delay Analysis

B.2.1 When Delay Occurs in WFQ

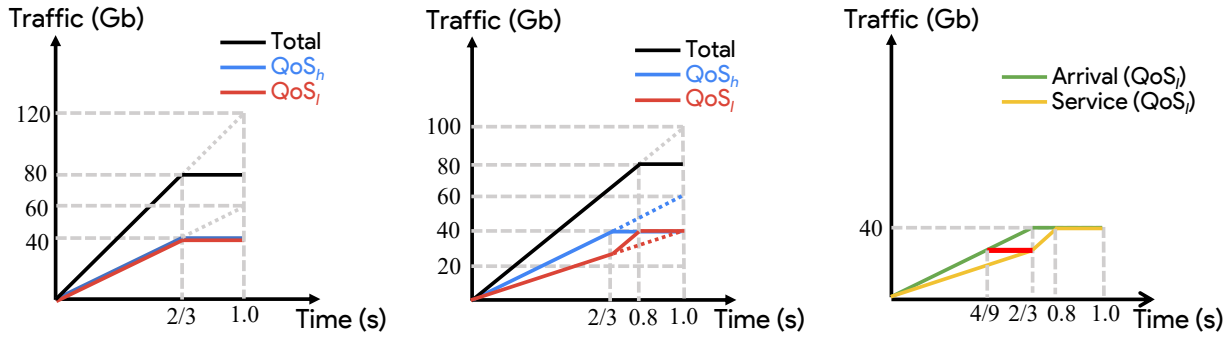
Delay occurs in a WFQ class *iff* (1) the total instantaneous input arrival rate on the link is greater than the link capacity (i.e., in the presence of overload), and (2) the arrival rate of the class is greater than the service rate of the class:

$$\sum a_i \ > r \ and \ a_i > s_i \tag{B.1}$$

where $a_i \geq s_i \geq min(a_i, g_i)$. Further, note that when the total arrival rate is greater than the link rate r, the total amount of service rate which WFQ can provide is fixed at r by the definition of work conservation:

$$\sum s_i = r, \ if \sum a_i > r \tag{B.2}$$

The expression for s_i is not immediately given as one needs to consider both QoS_i's own guaranteed rate g_i as well as a share of unused rate from other QoS classes due to the work conserving nature of WFQ.



(a) Arrival Curve (b) Service Curve (c) Delay Bound

Figure B.2: Applying network calculus on WFQ with 2 QoS levels.

Thus, the value s_i changes dynamically as the QoS-mix varies (e.g., when all the traffic from some QoS class has been consumed). In other words, QoS-mix in a WFQ can affect the delay profile of each QoS class. If the number of classes is 2, QoS_h and QoS_l, we can show that delay occurs on QoS_i if the following condition holds true:

$$\sum a_i > r \text{ and } a_i > g_i \tag{B.3}$$

We will show this for QoS_h, and QoS_l follows from symmetry. To prove this, we will show that in the two QoS case, if $a_h > s_h$, then $a_h > g_h$. We use the following equations for the proof:

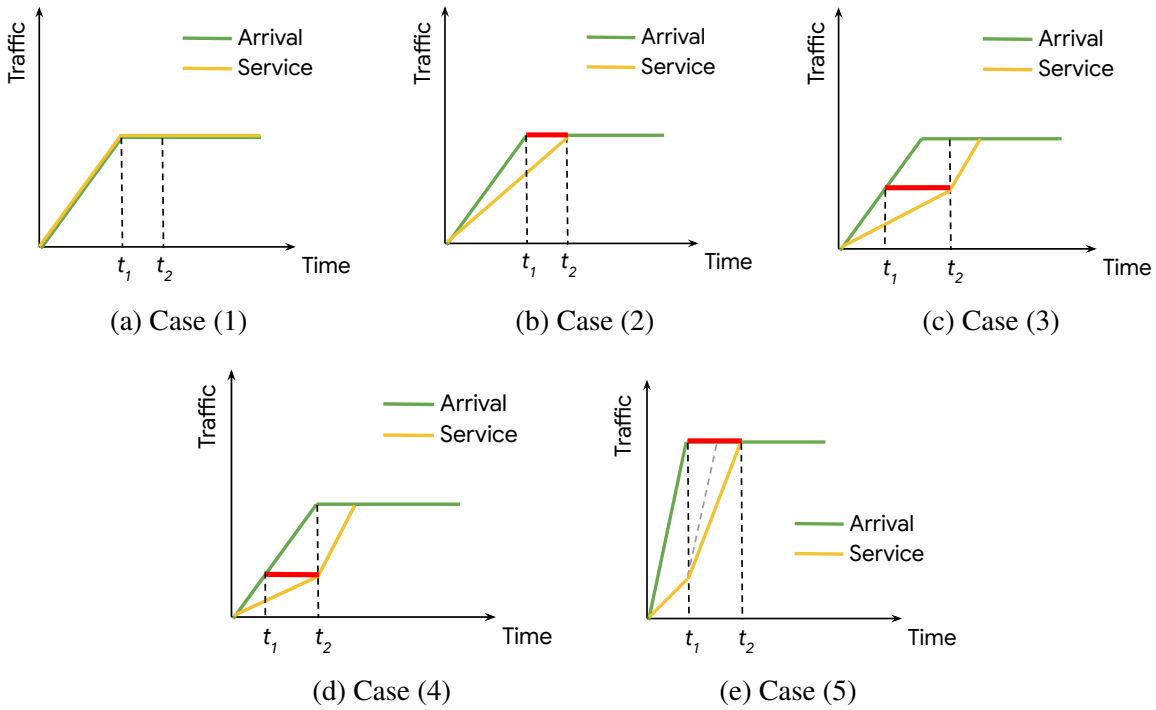
(1) *Overload Condition*: $a_h + a_l > r$
(2) *Work Conservation Condition*: $s_h + s_l = r$
(3) *WFQ Condition*: $g_h + g_l = r$
(4) *Guaranteed-bandwidth Condition*: If $a_i \geq g_i$, then $s_i \geq g_i$
(5) *Excess-bandwidth Condition*: If $s_h > g_h$, then $(a_h > g_h)$ and $(a_l < g_l)$

Consider the two possible scenarios depending on whether the arrival rate on QoS_l is below or above its guaranteed share.

Case 1: $a_l \leq g_l$ In this case, all of QoS_l traffic gets instantaneously served, i.e. $s_l = a_l$. Above equations give us: $a_h > r - a_l > r - g_l = g_h$.

Case 2: $a_l \geq g_l$ In this case, QoS_l at least gets its guaranteed rate, i.e., $s_l \geq g_l$. However, since there is no excess in QoS_h (as $a_h > s_h$), $s_l = g_l$, therefore $a_h > s_h = r - s_l = r - g_l = g_h$.

The same proof applies to class l due to symmetry.



(a) Case (1) (b) Case (2) (c) Case (3)

(d) Case (4) (e) Case (5)

Figure B.3: Service curve changes as QoS-share of QoS_h changes.

B.2.2 Derivation of Delay Equations in 2-QoS WFQ

We first demonstrate how we apply Network Calculus [58] to find closed-form equations for worst-case delay in WFQ for the 2-QoS case.

Network Calculus

Similar to previous work [176], our analysis is based on Network Calculus. Network Calculus furnishes a simple result: given an arrival cumulative curve as per the traffic arrival pattern and a service curve defining how traffic is served in the system, then the maximum horizontal distance between the arrival and the service curves gives the theoretical delay bound of the queuing system.

Toy Example

Consider a simple example with a single bottleneck with 2 QoS levels, QoS_h and QoS_l, under the same traffic pattern described in §3.4.1 with a weight ratio of 4:1, guaranteed rate for QoS_h is 80Gbps and is 20Gbps for QoS_l. The switch has a processing rate of 100 Gbps. The total Traffic is split into 50%/50% on QoS_h/QoS_l. Traffic arrives at a burst of 120Gbps for some time (>100Gbps), and then stays idle to achieve an average of 80% load (=80Gbps). Since 50% of the traffic is on QoS_h with a burst load of 120%, the switch immediately processes all QoS_h traffic incoming at a rate of 60Gbps (as its lower than the guaranteed rate of 80Gbps), and the remaining 40 Gbps goes to QoS_l, resulting in QoS_l's queue size growing at 20Gbps. Once all QoS_h traffic has been processed,

the link is now able to process QoS_l outstanding traffic at 100Gbsp and QoS_l's queue stops growing. In other words, at this moment QoS_l experiences the maximum delay at this moment – when all of QoS_h traffic has been processed. Figure B.2b summarizes the service curve for QoS_l traffic, and the maximum horizontal distance between these 2 curves gives the delay bound (shown as the red bar in Figure B.2c), which is represented as a fraction of the data sending period (including the burst and idle stage).

Closed-form Equations for 2-QoS case

With this explanation, we can start providing derivation of worst-case delay for QoS_h, $Delay_h$ in Equation 3.1, under the model depicted in Figure 3.7. The central idea is to determine the different service curves as QoS-mix changes, a and formulate equations for the delay bound based on the maximum horizontal distance between the arrival and the service curve.

The following notation in addition to Table 3.1 will be frequently used in the derivation.

$$
\begin{aligned}
a_h &= \rho r x \\
a_l &= \rho r (1 - x) \\
g_h &= \frac{\phi}{\phi + 1} r \\
g_l &= \frac{1}{\phi + 1} r
\end{aligned}
$$

where x is the QoS_h-share, and the ratio of QoS weights for QoS_h: QoS_l is $\phi : 1$.

As the value of QoS_h-share (x in the equations) varies from 0 to 1, we enter different domains with different service curves and we explain the cases below.

Case (1) When QoS_h-share just starts to increase from 0, QoS_h's instantaneous input arrival rate can be immediately processed by its minimum guaranteed rate, and thus, QoS_h experiences no delay, i.e., $Delay_h(x) = 0$. In other words, the arrival and service curves overlap under this case, $s_h = a_h$ (as depicted in Figure B.3a). The following conditions characterize this domain:

$$
\begin{aligned}
& a_h \leq g_h \\
\implies & x \leq \frac{\phi}{\phi + 1} \frac{1}{\rho}
\end{aligned}
$$

Case (2) As QoS_h-share continues to increase, QoS_h starts to experience delay as well but it still finishes earlier than QoS_l. Therefore, priority inversion will not happen yet and we are still in the

admissible region. This domain implies:

$$\begin{cases} a_h > g_h \\ a_l > g_l \\ \dfrac{\mu x}{g_h} < \dfrac{\mu(1-x)}{g_l} \end{cases}$$
$$\implies \frac{\phi}{\phi+1}\frac{1}{\rho} < x \leq \frac{\phi}{\phi+1}$$

We draw the arrival and service curve of QoS_h for Case (2) in Figure B.3b. The delay bound can be achieved by taking the maximum horizontal distance between the two curves, which is the difference between t_1 and t_2 in the figure. In this case, t_1 is as per the model in Figure 3.7, and t_2 is calculated as the time it takes to consume QoS_h's incoming traffic with $s_h = g_h$.

$$t_1 = \frac{\mu}{\rho}$$
$$t_2 = \frac{\mu r x}{g_h} = \mu x \frac{\phi+1}{\phi}$$
$$Delay_h(x) = t_2 - t_1 = \mu(\frac{\phi+1}{\phi}x - \frac{1}{\rho})$$

Case (3) QoS_h delay keeps growing as QoS_h-share increases, and eventually QoS_h finishes processing all the incoming traffic later than QoS_l, causing priority inversion. To solve for the domain in this case, we have:

$$\begin{cases} a_h > g_h \\ a_l > g_l \\ \dfrac{\mu x}{g_h} >= \dfrac{\mu(1-x)}{g_l} \end{cases}$$
$$\implies \frac{\phi}{\phi+1} < x \leq 1 - \frac{1}{\phi+1}\frac{1}{\rho}$$

The arrival and service curve of QoS_h in Case (3) is depicted in Figure B.3c. Similar to the last case, the delay is calculated by the difference of t_1 and t_2. By definition, t_2 in this case is the time QoS_l finishes processing its input traffic using g_l in the current period. In the interval $[0, t_2]$, $s_h = g_h$.

One can find t_1 by matching the y-value of the arrival and the service curves in Figure B.3c as:

$$
\begin{aligned}
& a_h t_1 = g_h t_2 \\
& t_2 = \frac{\mu r(1-x)}{g_l} = \mu(1-x)(\phi+1) \\
& t_1 = \frac{g_h t_2}{a_h} = \frac{\mu(1-x)\phi}{\rho x} \\
& Delay_h(x) = t_2 - t_1 = \mu(1-x)(\phi + 1 - \frac{\phi}{\rho\, x})
\end{aligned}
$$

Case (4) As QoS_l-share keeps decreasing with increasing x, QoS_l eventually experiences no delay, however QoS_h continues to experience delay. This is because we are in the overload situation where $\rho > 1$, there exists at least one QoS level that must experience delay. This implies:

$$
\begin{aligned}
& a_l < g_l \\
\Longrightarrow\ & x > 1 - \frac{1}{\phi+1}\frac{1}{\rho}
\end{aligned}
$$

Figure B.3d describes the arrival and service curve of QoS_h in this case. t_2 marks the time when QoS_l finishes servicing its traffic. t_1 can be calculated in a similar way as Case (3) by matching the y-values:

$$
\begin{aligned}
& t_2 = \frac{\mu}{\rho} \\
& t_1 = \frac{(r-a_l)t_2}{a_h} = \frac{\mu}{\rho^2}\frac{1-\rho(1-x)}{x} \\
& Delay_h(x) = \mu(\frac{1}{\rho} - \frac{1}{\rho^2})\frac{1}{x}
\end{aligned}
$$

Case (5) As QoS_h-share keeps increasing, its arriving rate eventually exceeds the line rate. This creates the final arrival and service curve profile as shown in Figure B.3e. The domain in this case is represented by:

$$
\begin{aligned}
& a_h > r \\
\Longrightarrow\ & x > \frac{1}{\rho}
\end{aligned}
$$

Note that t_1 in Figure B.3e is when QoS_l finishes. There are multiple ways to obtain the delay bound in this case. A simple one is to recognize that t_2 is the time it takes to consume all the

incoming traffic. To solve for the delay bound, we have:

$$t_1 = \frac{\mu}{\rho}$$
$$t_2 = \frac{\mu r}{r} = \mu$$
$$Delay_h(x) = t_2 - t_1 = \mu(1 - \frac{1}{\rho})$$

In summary, the five cases above have the following delay characteristics:

(1) QoS_h has no delay but QoS_l has delay
(2) both have delay but QoS_h finishes earlier
(3) both have delay but QoS_l finishes earlier
(4) QoS_h has delay but QoS_l does not
(5) QoS_h's arrival rate is above the link rate

By combining the above 5 cases, QoS_h worst-case delay ($Delay_h(x)$) (Equation 3.1) is obtained as:

$$\begin{cases} 0, & x \le \frac{\phi}{\phi+1}\frac{1}{\rho} \\ \mu(\frac{\phi+1}{\phi}x - \frac{1}{\rho}), & \frac{\phi}{\phi+1}\frac{1}{\rho} < x \le \frac{\phi}{\phi+1} \\ \mu(1-x)(\phi+1-\frac{\phi}{\rho\, x}), & \frac{\phi}{\phi+1} < x \le \min\{1 - \frac{1}{\phi+1}\frac{1}{\rho}, \frac{1}{\rho}\} \\ \mu(\frac{1}{\rho} - \frac{1}{\rho^2})\frac{1}{x}, & \min\{1 - \frac{1}{\phi+1}\frac{1}{\rho}, \frac{1}{\rho}\} < x \le \frac{1}{\rho} \\ \mu(1 - \frac{1}{\rho}), & x > \max\{\frac{\phi}{\phi+1}, \frac{1}{\rho}\} \end{cases}$$

Note that depending on the value of ρ and ϕ, it is possible that some of the domains are empty, however the equations remain valid with different domain boundaries.

For example, when $\phi = 4$, $\rho = 2$ and $\mu = 0.8$, QoS_h worst-case delay becomes:

$$Delay_h(x) = \begin{cases} 0, & x \le 0.4 \\ x - 0.4, & 0.4 < x \le 0.8 \\ 0.4, & x > 0.8 \end{cases}$$

Following a similar analysis, the closed-form worst-case delay for QoS_l ($Delay_l(x)$) is given

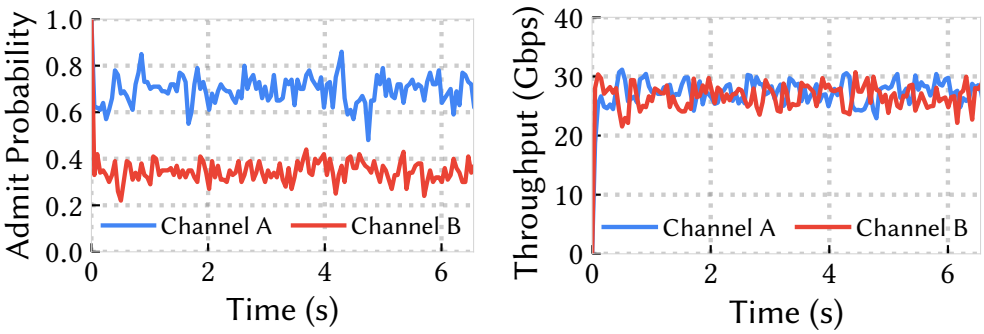


Figure B.4: Same experiment as in Figure 3.17 with smaller β value ($\beta = 0.0015$).

by:

$$\begin{cases} \mu(1-\frac{1}{\rho}), & x \leq \min\{1-\frac{1}{\rho}, \frac{\phi}{\phi+1}\} \\ \mu(\frac{1}{\rho}-\frac{1}{\rho^2})\frac{1}{(1-x)}, & 1-\frac{1}{\rho} < x \leq \max\{\frac{\phi}{\phi+1}\frac{1}{\rho}, 1-\frac{1}{\rho}\} \\ \mu\frac{x}{\phi}(\phi+1-\frac{1}{\rho(1-x)}), & \max\{\frac{\phi}{\phi+1}\frac{1}{\rho}, 1-\frac{1}{\rho}\} < x \leq \frac{\phi}{\phi+1} \\ \mu((\phi+1)(1-x)-\frac{1}{\rho}), & \frac{\phi}{\phi+1} < x \leq 1-\frac{1}{\phi+1}\frac{1}{\rho} \\ 0, & x > 1-\frac{1}{\phi+1}\frac{1}{\rho} \end{cases} \tag{B.4}$$

B.3 Sensitivity Analysis

α and β are key parameters in that they posit a tradeoff between SLO-compliance and stability in achieved shares. Under adversarial patterns, Aequitas favors SLO-compliance over work-conservation by the virtue of being an admission-control system.

We repeat the experiment in §3.6.5 with a lower β value of 0.0015 per MTU (Algorithm 2) compared to the original setting of 0.01. A lower β value implies a smaller decrease in p_{admit} whenever a latency miss is detected. While this provides excellent stability around fair-shares, it is less suited towards SLO-compliance. We show the equivalent of Figure 3.17 with smaller β value in Figure B.4, and the equivalent of Figure 3.18 in Figure B.5. The 1^{st}-p p_{admit} for RPC Channel A in Figure B.5 is 0.96, an improvement over 0.82 in Figure 3.18.

The parameter α has a similar tradeoff, a lower value is averse to increasing admit probability and favors SLO-compliance, whereas a higher value can provide better stability but with worse SLO-compliance.

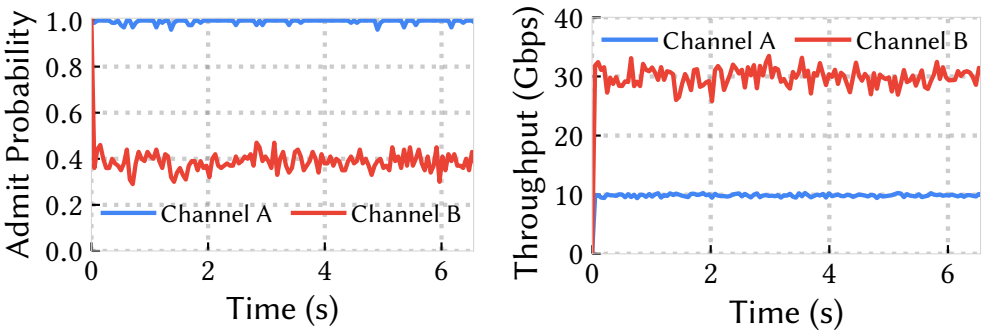


Figure B.5: Same experiment as in Figure 3.18 with smaller β value ($\beta = 0.0015$).

B.4 Artifact Appendix

We provide a brief appendix of our artifact for interested readers who want to try out Aequitas. You can find more information about our artifact evaluation on the *artifact-eval* branch of our GitHub repository (https://github.com/SymbioticLab/Aequitas).

B.4.1 Abstract

This artifact is designed to reproduce major results in this work using the simulator we wrote. It contains the source code of the simulator and the scripts used to launch experiments that reproduce our evaluation results.

B.4.2 Scope

The artifact is used to validate major evaluation results including theoretical 2-QoS worst-case delay analysis, SLO compliance, QoS-mix convergence, and fairness aspects of Aequitas. Readers are also encouraged to use the simulator or build on top of it to study other research problems in datacenter networking.

B.4.3 Contents

The artifact contains the following items:

(1) An README describing the artifact including how to build the simulator code, how to launch the selected experiments, and what expected results are after running those experiments.

(2) Source code of the simulator, which is used to simulate Aequitas and other systems in related work.

(3) Scripts to launch all the provided experiments.

The experiments include (a) a theoretical verification of the 2-QoS worst case delay, (b) evaluating SLO-compliance for both 2-QoS and 3-QoS cases, (c) evaluating QoS-mix convergence in a 3-QoS setting, and (d) evaluating fairness aspect in the same setting as the one we have in the chapter. We also includes the configuration files for all the systems in our related work comparison. Users can try out the simulator with their own RPC size distribution.

B.4.4 Hosting

To obtain the artifact, go to our GitHub repository at https://github.com/SymbioticLab/Aequitas and switch to the *artifact-eval* branch with the latest commit.

B.4.5 Requirements

The artifact requires a build environment to compile the simulator, which is written in C++. We developed the simulator in Linux. The code should work as long as it can be compiled correctly with automake. The compilation is very straightforward.

APPENDIX C

C.1 Additional Evaluation Results

C.1.1 Datasets and Query Configuration

We describe below the details of the datasets and query configuration we used in Vulvan.

Datasets:

The traffic monitoring queries use videos captured by traffic cameras among different metropolitan areas in Bellevue and Washington D.C. The videos are encoded in MP4 format (1280x720p, 30fps, and 120 seconds long) and randomly sampled from two-hour long videos among 24 days.

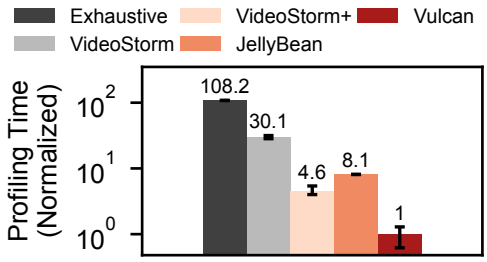
For autonomous driving perception queries, we use LiDAR sensor data from nuScenes [45], a large-scale autonomous driving dataset. The dataset features 1000 20-second driving scenes collected over months, in Boston and Singapore encompassing a diverse range of challenging driving situations.

Automatic speech recognition queries use the VOiCES dataset [186], an English speech audio dataset by male and female speakers. The dataset contains 3903 audio files (total 15 hours long) containing different room settings, simulated head movement, and various background noise patterns.

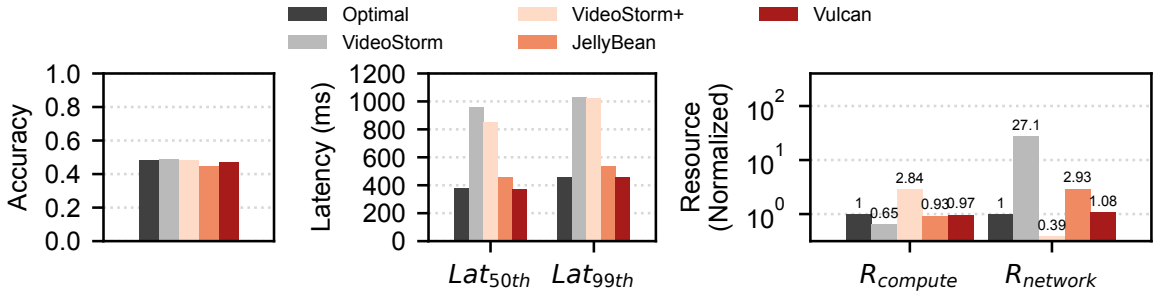
Configuration Knobs. Besides the variation of ML models described in §4.6.1, our queries in Evaluation configure the following additional configuration knobs.

- Video monitoring queries configure input frame sampling rate (1/2, 1/3, 1/4, 1/5, 1/6) and frame resizing factor (0.6, 0.7, 0.8, 0.9, 1) for frame resolution.
- Autonomous driving queris configure the ground removal factor[1] and voxel size, each with 5 configuration values (0.1, 0.2, 0.3, 0.4, 0.5).
- Speech recognition queires configure audio sampling rate (8k, 10k, 12k, 14k, 16k) and frequency mask width (500, 1000, 2000, 3000, 4000) of the noise reduction module.

[1] corresponds to the maximum distance between a ground point and the estimated 2D ground plane.



(a) Profiling Cost



(b) Performance and Resource Consumption

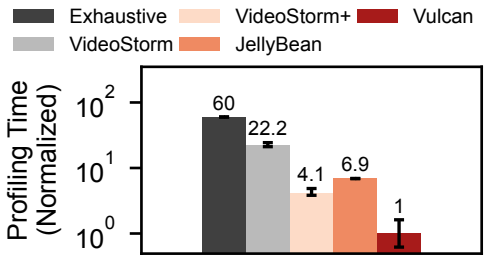
Figure C.1: End-to-end performance when exploring the best placement and query configuration for autonomous driving queries.

C.1.2 End-to-End Improvement

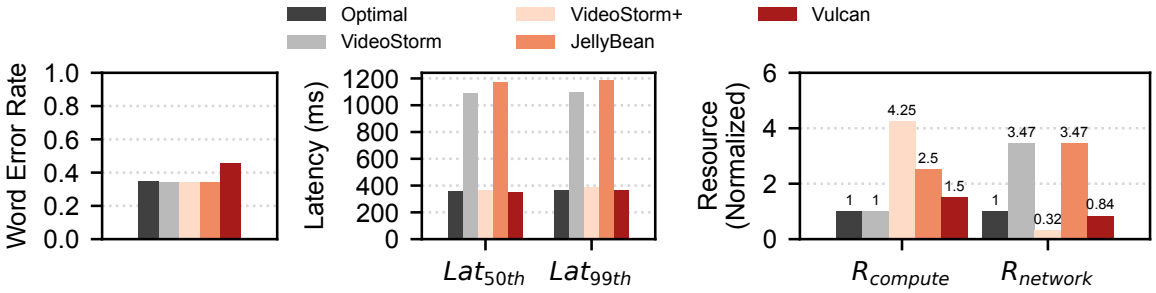
Figure C.1 and Figure C.2 shows the comparison between Vulcan and other baselines in the end-to-end performance of autonomous driving and speech recognition queries. The same conclusion can drawn as mentioned in §4.6.2.

C.1.3 Selecting Better Placement

Figure C.3 and Figure C.4 records the performance and resource consumption of autonomous driving and speech recognition queries, where PN, PC, and NC use the optimal query plan achieved by exhaustive search. Similar to §4.6.4, Vulcan always achieves the best placement choice, outperforming other baselines in query performance and resource consumption.



(a) Profiling Cost



(b) Performance and Resource Consumption

Figure C.2: End-to-end performance when exploring the best placement and query configuration for speech recognition queries.

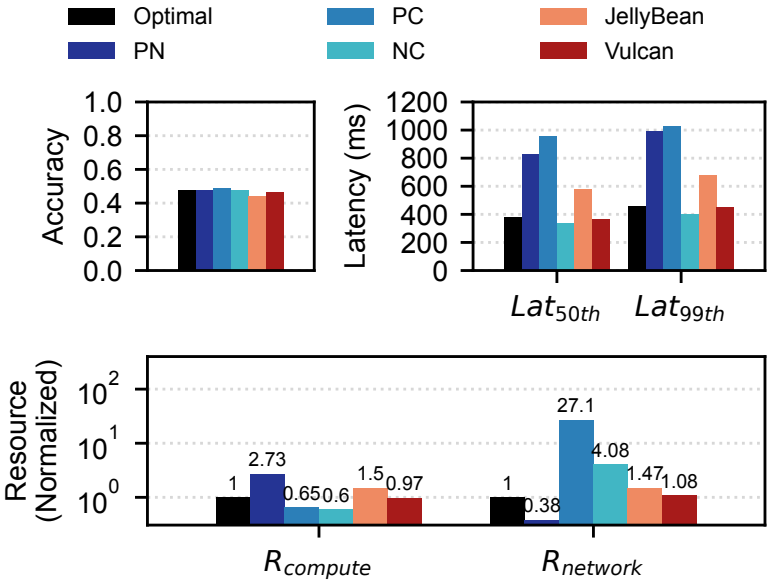


Figure C.3: Comparing Vulcan with different placement strategies on serving AD perception queries.

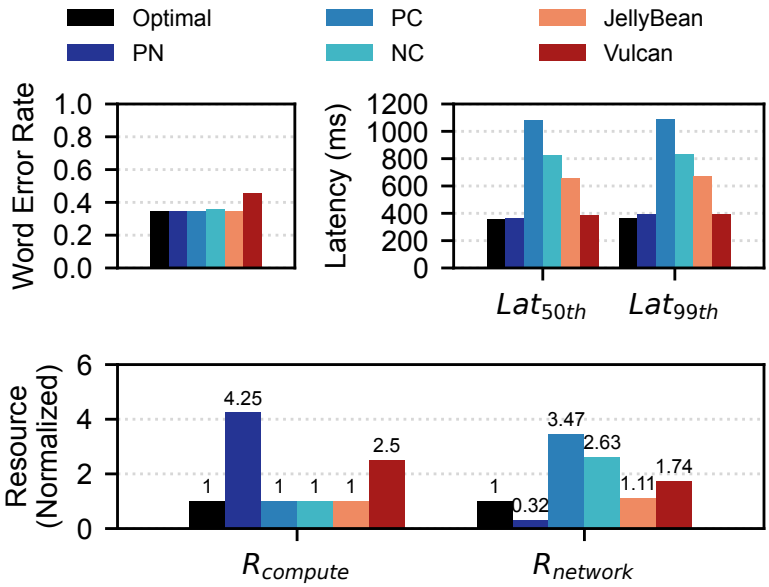


Figure C.4: Comparing Vulcan with different placement strategies on serving ASR queries.

BIBLIOGRAPHY

[1] RocksDB: A persistent key-value store for fast storage environments. https://rocksdb.org/.

[2] Numa balancing. Link to NUMA Balancing, 2012.

[3] memtier_benchmark: A high-throughput benchmarking tool for redis & memcached. Link to the benchmark, 2013.

[4] Linux cgroups. https://docs.kernel.org/admin-guide/cgroup-v2.html, 2015.

[5] Intel platform qos technologies. Link to PQoS, 2018.

[6] Microsoft Rocket for Live Video Analytics. Link to Microsoft Rocket, 2020.

[7] Azure Edge Zones with AT&T. Link to the blog post, 2022.

[8] Build modern connected applications at the edge with 5G. Link to the blog post, 2022.

[9] Edge video service (EVS). Link to EVS, 2022.

[10] Voci: Real-time speech recognition. https://www.vocitec.com/ads/real-time-speech-to-text, 2022.

[11] Yolov5. https://github.com/ultralytics/yolov5, 2022.

[12] Azure public multi-access edge compute (MEC). Link to Azure MEC, 2023.

[13] Kubernetes. https://github.com/kubernetes/kubernetes, 2023.

[14] Pebs. intel 64 and ia-32 architectures software developer's manual. https://software.intel.com/articles/intel-sdm/, 2023.

[15] Compute express link (cxl). https://www.computeexpresslink.org/, 2024.

[16] Inference of meta's llama model (and others) in pure c/c++. https://github.com/ggerganov/llama.cpp/, 2024.

[17] memcached. https://memcached.org/, 2024.

[18] Redis. https://github.com/redis/redis/, 2024.

[19] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *OSDI*, 2016.

[20] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *ASPLOS*, 2017.

[21] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. Sincronia: Near-optimal network design for coflows. In *SIGCOMM*, 2018.

[22] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, , and Michael Wei. Remote regions: a simple abstraction for remote memor. In *ATC*, 2018.

[23] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, 2017.

[24] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *SIGCOMM*, 2010.

[25] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick Mckeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal near-optimal datacenter transport. In *SIGCOMM*, 2013.

[26] AMD. μProf User Guide. 2024.

[27] Lisa Amini, Navendu Jain, Anshul Sehgal, Jeremy Silber, and Olivier Verscheure. Adaptive control of extreme-scale stream processing systems. In *ICDCS*, 2006.

[28] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end performance isolation through virtual datacenters. 2014.

[29] Apache. Apache crail. http://crail.incubator.apache.org/., 2021.

[30] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed nics. In *NSDI*, 2020.

[31] Infiniband Trade Association. Infiniband architecture specification volume 1. https://cw.infinibandta.org/document/dl/7859, 2015.

[32] Alexei Baevski, Henry Zhou, Abdelrahman Mohamed, and Michael Auli. wav2vec 2.0: A framework for self-supervised learning of speech representations, 2020.

[33] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *NSDI*, 2015.

[34] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, 2011.

[35] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. Chatty tenants and the cloud network sharing problem. In *NSDI*, 2013.

[36] Nikhil Bansal and Mor Harchol-Balter. Analysis of srpt scheduling: Investigating unfairness. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '01, page 279–290, New York, NY, USA, 2001. Association for Computing Machinery.

[37] Favyen Bastani, Songtao He, Arjun Balasingam, Karthik Gopalakrishnan, Mohammad Alizadeh, Hari Balakrishnan, Michael Cafarella, Tim Kraska, and Sam Madden. Miris: Fast object track queries in video. In *SIGMOD*, 2020.

[38] J.C.R. Bennett and H. Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *INFOCOM*, 1996.

[39] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. In *NSDI*, 2022.

[40] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, , and W. Weiss. Rfc2475: An architecture for differentiated service. In *IETF*, 1998.

[41] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. Rfc2475: An architecture for differentiated service. In *IETF*, 1998.

[42] Eric Brochu, Tyson Brochu, and Nando de Freitas. A bayesian interactive optimization approach to procedural animation design. In *SCA*, 2010.

[43] Eric Brochu, Vlad M. Cora, and Nando de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning, 2010.

[44] Eric Brochu, Matthew W. Hoffman, and Nando de Freitas. Portfolio allocation for bayesian optimization. In *UAI*, 2011.

[45] Holger Caesar, Varun Bankiti, Alex H Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuscenes: A multimodal dataset for autonomous driving. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11621–11631, 2020.

[46] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 79–92, 2021.

[47] Christopher Canel, Thomas Kim, Giulio Zhou, Conglong Li, Hyeontaek Lim, David G Andersen, Michael Kaminsky, and Subramanya Dulloor. Scaling video analytics on constrained edge nodes. In *MLSys*, 2019.

[48] Shuang Chen, Christina Delimitrou, and José F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *ASPLOS*, 2019.

[49] Dah-Ming Chiu and Raj Jain. Analysis of increase and decrease algorithms for congestion avoidance in computer networks. In *Computer Networks and ISDN systems*, 1989.

[50] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload control for µs-scale rpcs with breakwater. In *NSDI*, 2020.

[51] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. HUG: Multi-resource fairness for correlated and elastic demands. In *NSDI*, 2016.

[52] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *SIGCOMM*, 2015.

[53] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with Varys. In *SIGCOMM*, 2014.

[54] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *ATC*, 2017.

[55] Google Cloud. Speech-to-text: Automatic speech recognition. https://cloud.google.com/speech-to-text, 2022.

[56] Jeremy Cloud. Decomposing twitter: Adventures in serviceoriented architecture. In *n QCon New York*, 2013.

[57] Cloudlab. http://cloudlab.us/.

[58] Rene L. Cruz. A calculus for network delay, part i: Network elements in isolation. In *IEEE/ACM Transactions on Information Theory*, 1991.

[59] Rene L Cruz. Service burstiness and dynamic burstiness measures: A framework. *Journal of High Speed Networks*, 1(2):105–127, 1992.

[60] RL Cruz. A calculus for network delay, Part I: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.

[61] RL Cruz. A calculus for network delay, Part II: Network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, 1991.

[62] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, , and Amin Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *NSDI*, 2018.

[63] Matei Zaharia Daniel Kang, Peter Bailis. Blazeit: Optimizing declarative aggregation and limit queries for neural network-based video analytics. In *VLDB*, 2020.

[64] Nilaksh Das, Monica Sunkara, Dhanush Bekal, Duen Horng Chau, Sravan Bodapati, and Katrin Kirchhoff. Listen, know and spell: Knowledge-infused subword modeling for improving asr performance of oov named entities. In *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7887–7891, 2022.

[65] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM*, 1989.

[66] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 621–637, New York, NY, USA, 2021. Association for Computing Machinery.

[67] Fahad Dogar, Thomas Karagiannis, Hitesh Ballani, and Ant Rowstron. Decentralized task-aware scheduling for data center networks. In *SIGCOMM*, 2014.

[68] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. 2024.

[69] Aleksandar Dragojevic, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast remote memory. In *NSDI*, 2014.

[70] Paul J. Drongowski. Instruction-based sampling: A new performance analysis technique for amd family 10h processors. 2007.

[71] Nick G. Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, Kadangode K Ramakrishnan, and Jacobus E van der Merwe. A flexible model for resource management in virtual private networks. In *SIGCOMM*, 1999.

[72] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Chris Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *ASPLOS*, 2023.

[73] Amazon DynamoDB. https://aws.amazon.com/dynamodb/.

[74] John Emmons, Sadjad Fouladi, Ganesh Ananthanarayanan, Shivaram Venkataraman, Silvio Savarese, and Keith Winstein. Cracking open the dnn black-box: Video analytics with dnns across the camera-cloud boundary. In *HotEdgeVideo*, 2019.

[75] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An infrastructure for inline acceleration of network applications. In *USENIX ATC*, 2019.

[76] Abdullah Bin Faisal, Hafiz Mohsin Bashir, Ihsan Ayyub Qazi, Zartash Uzmi, and Fahad R. Dogar. Workload adaptive flow scheduling. In *CoNEXT*, 2018.

[77] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, , and Albert Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *NSDI*, 2018.

[78] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transaction on Networking*, 1(4):397–1413, 1993.

[79] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *OSDI*, 2020.

[80] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *CoNEXT*, 2015.

[81] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. When cloud storage meets {RDMA}. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533, 2021.

[82] Alireza Ghasemieh and Rasha Kashef. 3d object detection for autonomous driving: Methods, models, sensors, data, and challenges. *Transportation Engineering*, 8:100115, 2022.

[83] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. 2012.

[84] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[85] S. Jamaloddin Golestani. Network delay analysis of a class of fair queueing algorithms. *IEEE JSAC*, 13(6):1057–1070, 1995.

[86] Tiago Gomes, Diogo Matias, André Campos, Luís Cunha, and Ricardo Roriz. A survey on ground segmentation methods for automotive lidar sensors. *Sensors*, 23(2), 2023.

[87] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access,{High-Performance} memory disaggregation with {DirectCXL}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, 2022.

[88] Pawan Goyal, Harrick M Vin, and Haichen Chen. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *SIGCOMM*, 1996.

[89] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can jump them! In *NSDI*, 2015.

[90] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with Infiniswap. In *NSDI*, 2017.

[91] Chuanxiong Guo, Guohan Lu, Helen J Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. SecondNet: A data center network virtualization architecture with bandwidth guarantees. In *CoNEXT*, 2010.

[92] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity Ethernet at scale. In *SIGCOMM*, 2016.

[93] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *SIGCOMM*, 2015.

[94] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *MobiSys*, 2016.

[95] Daniel N Hill, Houssam Nassif, Yi Liu, Anand Iyer, and S V N Vishwanathan. An efficient bandit algorithm for realtime multivariate optimization. In *KDD*, 2017.

[96] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing flows quickly with preemptive scheduling. In *SIGCOMM*, 2012.

[97] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *SIGCOMM*, 2013.

[98] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *OSDI*, 2018.

[99] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in Windows Azure Storage. In *USENIX ATC*, 2012.

[100] Intel. HTB Home. http://luxik.cdi.cz/~devik/qos/htb/, 2003.

[101] Van Jacobson and Michael J. Karels. Congestion avoidance and control. In *SIGCOMM*, 1988.

[102] Jeffrey M Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, 29(7):954–962, 1981.

[103] Samvit Jain, Xun Zhang, Yuhao Zhou, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Paramvir Bahl, and Joseph Gonzalez. Spatula: Efficient Cross-camera Video Analytics on Large Camera Networks. In *ACM/IEEE Symposium on Edge Computing (SEC)*, 2020.

[104] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable message latency in the cloud. In *SIGCOMM*, 2015.

[105] Olivier Chapelle Jean-Baptiste Tien, joycenv. Display advertising challenge, 2014.

[106] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazieres, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. EyeQ: Practical network performance isolation at the edge. In *NSDI*, 2013.

[107] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: Scalable adaptation of video analytics. In *SIGCOMM*, 2018.

[108] Ramesh Johari and John N. Tsitsiklis. Efficiency loss in a network resource allocation game. *Mathematics of Operations Research*, pages 29(3):407–435, 2004.

[109] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, page 13(4):455–492, 1998.

[110] Kai Kai Jüngling and Michael Arens. Local feature based person reidentification in infrared image sequences. In *IEEE AVSS*, 2010.

[111] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *SIGCOMM*, 2014.

[112] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *OSDI*, 2016.

[113] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter rpcs can be general and fast. In *NSDI*, 2019.

[114] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: Optimizing neural network queries over video at scale. In *PVLDB*, 2017.

[115] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *ASPLOS*, 2017.

[116] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. In *ISCA*, 2017.

[117] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *ASPLOS*, 2016.

[118] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *EuroSys*, 2019.

[119] Mehrdad Khani, Ganesh Ananthanarayanan, Kevin Hsieh, Junchen Jiang, Ravi Netravali, Yuanchao Shu, Mohammad Alizadeh, and Victor Bahl. Recl: Responsive resource-efficient continuous learning for video analytics. In *NSDI*, 2023.

[120] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. Freeflow: Software-based virtual RDMA networking for containerized clouds. In *NSDI*, 2019.

[121] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for multi-tiered memory systems. In *ATC*, 2021.

[122] Yuki Koyama, Issei Sato, Daisuke Sakamoto, and Takeo Igarashi. Sequential line search for efficient visual design optimization by crowds. In *ACM Transactions on Graphics*, 2017.

[123] Brian Kulis, Prateek Jain, and Kristen Grauman. Fast similarity search for learned metrics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(12):2143–2157, 2009.

[124] Alok Kumar, Sushant Jain, Uday Naik, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *SIGCOMM*, 2015.

[125] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *SIGCOMM*, 2020.

[126] Gautam Kumar, Srikanth Kandula, Peter Bodik, and Ishai Menache. Virtualizing traffic shapers for practical resource allocation. In *HotCloud*, 2013.

[127] Gautam Kumar, Akshay Narayan, and Peter Gao. Yaps: Yet another packet simulator. https://github.com/NetSys/simulator, 2016.

[128] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, and Amin Vahdat. Picnic: predictable virtualized nic. In *SIGCOMM*, 2019.

[129] Harold J. Kushner. A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. *Journal of Basic Engineering*, page 86:97–106, 1964.

[130] Alex H Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. Pointpillars: Fast encoders for object detection from point clouds. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12697–12705, 2019.

[131] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang Aditya Akella, Michael M. Swift, and T.V. Lakshman. Uno: Unifying host and smart nic offload for flexible packet processing. In *SoCC*, 2017.

[132] Yanfang Le, Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael M. Swift. RoGUE: RDMA over generic unconverged ethernet. In *SoCC*, 2018.

[133] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *ASPLOS*, 2023.

[134] Wei Li, Rui Zhao, Tong Xiao, and Xiaogang Wang. Deepreid: Deep filter pairing neural network for person re-identification. In *CVPR*, 2014.

[135] Yuliang LI, Harry Hongqiang Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *SIGCOMM*, 2019.

[136] Zhuqi Li, Yuanchao Shu, Ganesh Ananthanarayanan, Longfei Shangguan, Kyle Jamieson, and Paramvir Bahl. Spider: A Multi-Hop Millimeter-Wave Network for Live Video Analytics. In *ACM/IEEE Symposium on Edge Computing (SEC)*, 2021.

[137] Giuseppe Lisanti, Iacopo Masi, Andrew D. Bagdanov, and Alberto Del Bimbo. Person re-identification by iterative re-weighted sparse ranking. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2014.

[138] John D. C. Little. A proof for the queuing formula: L=λw. In *Operations Research*, 1961.

[139] Linda Liu, Yi Gu, Aditya Gourav, Ankur Gandhe, Shashank Kalmane, Denis Filimonov, Ariya Rastrow, and Ivan Bulyko. Domain-aware neural language models for speech recognition. In *ICASSP 2021*, 2021.

[140] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *SIGCOMM*, 2019.

[141] Daniel Lizotte, Tao Wang, Michael Bowling, and Dale Schuurmans. Automatic gait optimization with gaussian process regression. In *IJCAI*, 2007.

[142] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *ISCA*, 2015.

[143] Franz Loewenherz. Video analytics towards vision zero. Link to the presentation, 2017.

[144] Chenglang Lu, Mingyong Liu, and Zongda Wu. Svql: A sql extended query language for video databases. In *IJDTA*, 2015.

[145] Yan Lu, Shiqi Jiang, Ting Cao, and Yuanchao Shu. Turbo: Opportunistic Enhancement for Edge Video Analytics. In *ACM Conference on Embedded Network Sensor Systems (SenSys)*, 2022.

[146] Yan Lu, Zhun Zhong, and Yuanchao Shu. Multi-View Domain Adaptive Object Detection in Surveillance Cameras. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2023.

[147] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. Accelerating machine learning inference with probabilistic predicates. In *SIGMOD*, 2018.

[148] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[149] Ruben Martinez-Cantin, Nando de Freitas, Eric Brochu, Jose Castellanos, and Arnaud Doucet. A bayesian exploration-exploitation approach for optimal online sensing and planning with a visually guided mobile robot. *Autonomous Robots*, pages 27(2):93–103, 2009.

[150] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *SOSP*, 2019.

[151] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent page placement for CXL-enabled tiered memory. In *ASPLOS*, 2023.

[152] Tony Mauro. Adopting microservices at netflix: Lessons for architectural design. https://tinyurl.com/htfezlj, 2015.

[153] Massimo Merenda, Carlo Porcaro, and Demetrio Iero. Edge machine learning for ai-enabled iot devices: A review. *Sensors*, 20(9), 2020.

[154] Robert C. Merton. *Continuous-Time Finance*. Blackwell, 1990.

[155] Meta. Llama 2 70b - gguf model, 2024.

[156] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based congestion control for the datacenter. In *SIGCOMM*, 2015.

[157] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for RDMA. In *SIGCOMM*, 2018.

[158] Jonas Mockus. *Bayesian Approach to Global Optimization.* Kluwer Academic, 1989.

[159] Jeffrey C Mogul and Lucian Popa. What we talk about when we talk about cloud network performance. *SIGCOMM CCR*, 42(5):44–48, 2012.

[160] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM*, 2018.

[161] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 374–385, 2011.

[162] Kanthi Nagaraj, Dinesh Bharadia, Hongzi Mao, Sandeep Chinchali, Mohammad Alizadeh, and Sachin Katti. NUMFabric: Fast and flexible bandwidth allocation in datacenters. In *SIGCOMM*, 2016.

[163] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.

[164] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *USENIX ATC*, 2015.

[165] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. Fair queuing memory systems. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 208–222, 2006.

[166] Yuanjiang Ni, Pankaj Mehra, Ethan Miller, and Heiner Litz. Tmc: Near-optimal resource allocation for tiered-memory systems. In *SoCC*, 2023.

[167] Kathleen Nichols and Van Jacobson. Controlling queue delay: A modern aqm is just one piece of the solution to bufferbloat. In *ACM Queue*, 2012.

[168] Shadi Noghabi, Landon Cox, Sharad Agarwal, and Ganesh Ananthanarayanan. The emerging landscape of edge-computing. In *ACM SIGMOBILE GetMobile*, 2020.

[169] Pirouz Nourian, Romulo Gonçalves, Sisi Zlatanova, Ken Arroyo Ohori, and Anh Vu Vo. Voxelization algorithms for geospatial applications: Computational methods for voxelating spatial datasets of 3d city models containing 3d surface, curve and point data models. *MethodsX*, 3:69–86, 2016.

[170] Oracle. Using persistent memory database. 2022.

[171] Amy Ousterhout, Jonathan Perry, Hari Balakrishnan, and Petr Lapukhov. Flexplane: An experimentation platform for resource management in datacenters. In *NSDI*, 2017.

[172] Arthi Padmanabhan, Neil Agarwal, Anand Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Guoqing Harry Xu, and Ravi Netravali. GEMEL: Model Merging for Memory-Efficient, Real-Time Video Analytics at the Edge. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[173] Dan Paik. Adapt or die: A microservices story at google. https://www.slideshare.net/apigee/adapt-or-die-a-microservices-story-at-google, 2016.

[174] Rong Pan, Lee Breslau, Balaji Prabhakar, and Scott Shenker. Approximate fairness through differential dropping. In *SIGCOMM*, 2003.

[175] Rong Pan, Balaji Prabhakar, and Konstantinos Psounis. Choke: A stateless active queue management scheme for approximating fair bandwidth allocation. In *INFOCOM*, 2000.

[176] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. In *IEEE/ACM Transactions on Networking*, 1993.

[177] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized zero-queue datacenter network. 2014.

[178] Marius Poke and Torsten Hoefler. DARE: High-performance state machine replication on rdma networks. In *HPDC*, 2015.

[179] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the network in cloud computing. In *SIGCOMM*, 2012.

[180] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. ElasticSwitch: Practical work-conserving bandwidth guarantees for cloud computing. In *SIGCOMM*, 2013.

[181] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[182] Hang Qiu, Pohan Huang, Namo Asavisanu, Xiaochen Liu, Konstantinos Psounis, and Ramesh Govindan. Autocast: Scalable infrastructure-less cooperative perception for distributed collaborative driving. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '22, 2022.

[183] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825. USENIX Association, November 2020.

[184] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, , and Simon Peter. TPP: Transparent page placement for CXL-enabled tiered memory. In *ASPLOS*, 2023.

[185] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. In *CVPR*, 2017.

[186] Colleen Richey, Maria A. Barrios, Zeb Armstrong, Chris Bartels, Horacio Franco, Martin Graciarena, Aaron Lawson, Mahesh Kumar Nandwana, Allen Stauffer, Julien van Hout, Paul Gamble, Jeff Hetherly, Cory Stephenson, and Karl Ni. Voices obscured in complex environmental settings (voices) corpus, 2018.

[187] Van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, 1979.

[188] Francisco Romero, Johann Hauswald, Aditi Partap, Daniel Kang, Matei Zaharia, and Christos Kozyrakis. Optimizing video analytics with declarative model relationships. *Proc. VLDB Endow.*, 16(3):447–460, nov 2022.

[189] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *SoCC*, 2021.

[190] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Carlo Contavalli, Amin Vahdat, et al. Carousel: Scalable traffic shaping at end hosts. In *SIGCOMM*, 2017.

[191] Hanrijanto Sariowan, Rene L. Cruz, and George C. Polyzos. Sced: A generalized scheduling policy for guaranteeing quality-of-service. In *IEEE/ACM Transactions on Networking*, 1999.

[192] Cristian Satnic. Amazon, microservices and the birth of aws cloud computing. Link to the post, 2016.

[193] sBrian T. Ratchford. Cost-benefit models for explaining consumer choice and information seeking behavior. *Management Science*, 28, 1982.

[194] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. vtmm: Tiered memory management for virtual machines. In *EuroSys*, 2023.

[195] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed os for hardware resource disaggregation. In *OSDI*, 2018.

[196] Shuyao Shi, Jiahe Cui, Zhehao Jiang, Zhenyu Yan, Guoliang Xing, Jianwei Niu, and Zhenchao Ouyang. Vips: Real-time perception fusion for infrastructure-assisted autonomous driving. In *MobiCom*, 2021.

[197] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Sharing the data center network. In *NSDI*, 2011.

[198] Jiang Shiqi, Lin Zhiqi, Li Yuanchun, Shu Yuanchao, and Liu Yunxin. Flexible High-resolution Object Detection on Edge Devices with Tunable Latency. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2021.

[199] Madhavapeddi Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *SIGCOMM*, 1995.

[200] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. 2012.

[201] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[202] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *NIPS*, 2012.

[203] SR-IOV. Single root i/o virtualization. http://pcisig.com/specifications/iov/single_root/., 2018.

[204] Niranjan Srinivas, Andreas Krause, Sham M. Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. In *ICML*, 2010.

[205] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and efficient nic packet scheduling. In *NSDI*, 2017.

[206] Brent Stephens, Aditya Akella, and Michael Swift. Your programmable nic should be a programmable switch. In *HotNets*, 2018.

[207] Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael Swift. Titan: Fair packet scheduling for commodity multiqueue nics. In *USENIX ATC*, 2017.

[208] Ion Stoica, Hui Zhang, and TS Ng. A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services. In *SIGCOMM*, 1997.

[209] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. Unification of temporary storage in the nodekernel architecture. In *ATC*, 2019.

[210] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *MICRO*, 2015.

[211] Mellanox Technologies. Mellanox Perftest Package. https://community.mellanox.com/docs/DOC-2802, 2017.

[212] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Ant Rowstron, Tom Talepy, Richard Black, and Timothy Zhu. Ioflow: A software-defined storage architecture. In *SOSP*, 2013.

[213] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[214] Shin-Yeh Tsai and Yiying Zhang. LITE kernel rdma support for datacenter applications. In *SOSP*, 2017.

[215] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: enabling high-level slos on shared storage systems. In *SoCC*, 2012.

[216] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A {Memory-Disaggregated} managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280, 2020.

[217] Xiao-Dong Wang, Xiao Chen, Jie Min, and Yu Zhou. A priority-based weighted fair queueing algorithm in wireless sensor network. In *WiCom*, 2012.

[218] Matt Welsh and David Culler. Overload management as a fundamental service design primitive. In *SIGOPS*, 2002.

[219] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowstron. Better never than late: Meeting deadlines in datacenter networks. In *SIGCOMM*, 2011.

[220] J. Wroclawski. Rfc 2210: The use of rsvp with ietf integrated services. In *IETF*, 1997.

[221] Yongji Wu, Matthew Lentz, Danyang Zhuo, and Yao Lu. Serving and optimizing machine learning workflows on heterogeneous infrastructures. In *VLDB*, 2023.

[222] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *ASPLOS*, 2023.

[223] Tianwei Yin, Xingyi Zhou, and Philipp Krahenbuhl. Center-based 3d object detection and tracking. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11784–11793, 2021.

[224] Ji Won Yoon, Beom Jun Woo, and Nam Soo Kim. Hubert-ee: Early exiting hubert for efficient speech recognition, 2022.

[225] David Zats, Anand Padmanabha Iyer, Ganesh Anantharayanan, Rachit Agarwal, Randy Katz, Ion Stoica, and Amin Vahdat. Fastlane: Making short flows shorter with agile drop notification. In *SOCC*, 2015.

[226] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *NSDI*, 2017.

[227] Xumiao Zhang, Anlan Zhang, Jiachen Sun, Xiao Zhu, Y. Ethan Guo, Feng Qian, and Z. Morley Mao. Emp: Edge-assisted multi-vehicle perception. In *MobiCom*, 2021.

[228] Xumiao Zhang, Anlan Zhang, Jiachen Sun, Xiao Zhu, Y. Ethan Guo, Feng Qian, and Z. Morley Mao. Emp: Edge-assisted multi-vehicle perception. In *MobiCom*, 2021.

[229] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. Aequitas: Admission control for performance-critical rpcs in datacenters. In *SIGCOMM*, 2022.

[230] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. Justitia: Software multi-tenancy in hardware kernel-bypass networks. In *NSDI*, 2022.

[231] Yiwen Zhang, Xumiao Zhang, Ganesh Ananthanarayanan, Anand Iyer, Yuanchao Shu, Victor Bahl, Morley Mao, and Mosharaf Chowdhury. Vulcan: Automatic query planning for live ml analytics. In *NSDI*, 2024.

[232] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junf. Overload control for scaling wechat microservices. In *SoCC*, 2018.

[233] Timothy Zhu, Daniel S. Berger, and Mor Harchol-Balter. Snc-meister: Admitting more tenants with tail latency slos. In *SoCC*, 2016.

[234] Timothy Zhu, Michael A. Kozuch, and Mor Harchol-Balter. Workloadcompactor: Reducing datacenter cost while providing tail latency slo guarantees. In *SoCC*, 2017.

[235] Timothy Zhu, Alexey Tumanov, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Prioritymeister: Tail latency qos for shared networked storage. In *SoCC*, 2014.

[236] Xinge Zhu, Yuexin Ma, Tai Wang, Yan Xu, Jianping Shi, and Dahua Lin. Ssn: Shape signature networks for multi-class object detection from point clouds. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXV 16*, pages 581–597. Springer, 2020.

[237] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In *SIGCOMM*, 2015.

[238] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In *SIGCOMM*, 2015.