Meeting tail latency SLOs in shared networked storage

Timothy Zhu

CMU-CS-17-105
May 2017

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the ISTC-CC, a Google Faculty Research Award 2015/16, and the National Science Foundation under grant numbers CNS-1116282, CCMI-1334194, CCF-1629444, and DGE-1252522.

Keywords: Resource Management of Computer Systems, Tail Latency, Quality of Service, Multi-tenancy, Cloud Computing, Storage Systems, Networks, Performance Modeling, Deterministic Network Calculus, Stochastic Network Calculus

For my family and friends, for supporting me in my endeavors.

Abstract

Shared computing infrastructures (e.g., cloud computing, enterprise datacenters) have become the norm today due to their lower operational costs and IT management costs. However, resource sharing introduces challenges in controlling performance for each of the workloads using the infrastructure. For user-facing workloads (e.g., web server, email server), one of the most important performance metrics companies want to control is tail latency, the time it takes to complete the most delayed requests. Ideally, companies would be able to specify tail latency performance goals, also called Service Level Objectives (SLOs), to ensure that almost all requests complete quickly.

Meeting tail latency SLOs is challenging for multiple reasons. First, tail latency is significantly affected by the burstiness that is commonly exhibited by production workloads. Burstiness leads to transient queueing, which is a major cause of high tail latency. Second, tail latency is often due to I/O (e.g., storage, networks), and I/O devices exhibit performance peculiarities that make it hard to meet SLOs. Third, the end-to-end latency is affected by sum of latencies across multiple types of resources such as storage and networks. Most of the existing research, however, have ignored burstiness and focused on a single resource.

This thesis introduces new techniques for meeting end-to-end tail latency SLOs in both storage and networks while accounting for the burstiness that arises in production workloads. We address open questions in scheduling policies, admission control, and workload placement. We build a new Quality of Service (QoS) system for meeting tail latency SLOs in networked storage infrastructures. Our system uses prioritization and rate limiting as tools for controlling the congestion between workloads. We introduce a novel approach for intelligently configuring the workload priorities and rate limits using two different types of queueing analyses: Deterministic Network Calculus (DNC) and Stochastic Network Calculus (SNC). By integrating these mathematical analyses into our system, we are able to build better algorithms for optimizing the resource usage. Our implementation results using realistic workload traces on a physical cluster demonstrate that our approach can meet tail latency SLOs while achieving better resource utilization than the state-of-the-art.

While this thesis focuses on scheduling policies, admission control, and workload placement in storage and networks, the ideas presented in our work can be applied to other related problems such as workload migration and datacenter provisioning. Our theoretically grounded techniques for controlling tail latency can also be extended beyond storage and networks to other contexts such as the CPU, cache, etc. For example, in real-time CPU scheduling contexts, our DNC-based techniques could be used to provide strict latency guarantees while accounting for workload burstiness.

Acknowledgments

Many people have played an important role in helping me succeed in my PhD program. First and foremost, I would like to thank my advisor, Mor Harchol-Balter, for all her time and effort in training me as a researcher. I have learned tremendously from her, and I'm deeply appreciative of her care and support. She has far exceeded my expectations of an advisor. I'd also like to thank the rest of my thesis committee (Greg Ganger, Dave Andersen, Mike Kozuch, Arif Merchant) for helping me shape this thesis. In particular, I'd like to especially thank Mike Kozuch, who has collaborated with me in nearly all of my research. He has been like a second advisor to me. Needless to say, I'm also very grateful to my other colleagues (Anshul Gandhi, Alexey Tumanov, Greg Ganger, Daniel Berger, Kristy Gardner, Ben Berg), my internship mentors (Eno Thereska, John Wilkes), and the awesome administrative staff at CMU (Deb Cavlovich, Karen Lindenfelser, Nancy Conway). Without all these amazing people, I would not be where I am today.

I'm fortunate to have friends and family who have greatly supported me during my PhD. Hilary Moyes, Rich Newby, Greg Newby, Hee Yeon Shin, Alex Newby, and Colette Newby have been a second family to me in Pittsburgh. Andy Echenique, Brad Yates, Rohit Ramnath, Catherine Hueston, Jon Chastek, and Johanna Chastek have been longtime friends who have always been there for me. I'd like to thank them all for their friendship over the years and the joy they've brought into my life. Finally, I would like to thank my family for their continual love and support. I'm eternally grateful to them for their care throughout my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computing has returned to the age of sharing resources, both for the masses (e.g., cloud computing) and enterprises (e.g., Google datacenters). This is because running on shared infrastructures significantly reduces both operational costs and IT management costs. However, sharing also introduces a myriad of challenges. How does one ensure performance goals for applications sharing resources? Which shared resources should be assigned to an application? How much performance isolation does one want between applications?

One of the hardest challenges in sharing resources is meeting *latency* performance goals. Latency is temporal and is highly affected by how requests queue in the system. Even if system utilization is low, latency can be poor if applications send requests at the same time. In the average case, the probability that applications send requests at the same time may be low enough. However, the *tail latency* (i.e., latency of the slowest requests) is still significantly impacted by queueing within the system.

In this thesis, we look at the question of how to meet tail latency Service Level Objectives (SLOs) (i.e., performance goals) when sharing networked storage systems in datacenters.

1.1 Motivation

Our work is motivated by three trends:

First, companies like Google and Amazon are growing concerned about long latencies at the 99th and 99.9th tail percentiles [20, 21]. As technology improves, users become accustomed to low latency and start to expect near instant response times. If one out of every hundred requests is slow, users may eventually switch to a competitor's product. Furthermore, as applications send more requests in parallel, the need for low tail latencies becomes increasingly important since jobs often run at the speed of the slowest request.

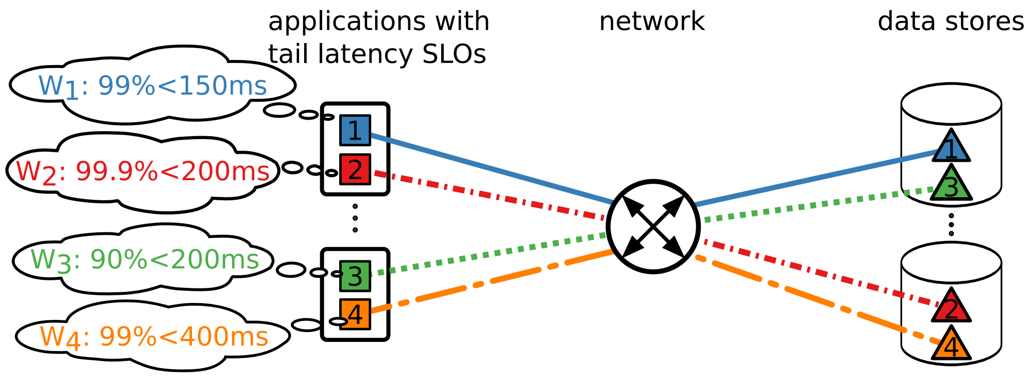


Figure 1.1: Our system operates in a shared networked storage environment within a datacenter. Applications (squares) run in client VMs and access storage volumes (triangles) on servers over a network. Each workload specifies a tail latency SLO, e.g., 99% of workload W_1's requests should have a latency less than 150ms ($99\% < 150ms$). As illustrated by the network paths, workloads may congest at different parts in the system. For example, workload W_1 and W_2 congest at the top left link, whereas W_1 and W_3 congest at the top right.

Second, with the growth in data-driven applications, I/O latencies due to storage and networks play a large part in the end-to-end user experience for latency sensitive applications. Storage is often the hardest resource to share and is typically the bottleneck resource. Unless storage accesses can be completely avoided, storage latencies typically have the most impact on tail latency.

Third, applications are moving into multi-tenant datacenter environments where resources are shared, particularly network and storage. This shift in industry to consolidate applications onto shared infrastructures is beneficial in reducing resource and IT management costs. However, while consolidation leads to greater economies of scale, it also introduces challenges in meeting tail latency SLOs.

1.2 Problem definition

Our work targets a networked storage environment within a datacenter, as illustrated in Figure 1.1. Our system is designed to supplement existing systems with the ability to provide tail latency guarantees. For example, our work could be applied to cloud storage settings such as Amazon's Elastic Block Store (EBS) or OpenStack's Cinder. It can also apply to enterprise settings with storage servers running, for example, Network File System (NFS).

We now define some common terminology used throughout this thesis:

Definition 1. (Application)

Applications (squares in Figure 1.1) run in client VMs and access storage volumes (triangles) on storage servers over a network. In our work, we focus on user-facing applications such as email servers or web servers that require low latency.

Definition 2. (Request)

Storage *requests* are sent over time by applications to access their data. A request is sent from a client to a server and back. Requests are represented by a timestamp, request type (read/write), request size (e.g., 4KB), and request offset (i.e., logical block address (LBA)).

Definition 3. (Workload)

An application's *workload* consists of a sequence of requests that arrive over time.

Definition 4. (Trace)

A storage *trace* is a list of requests with their timestamps, request types, request sizes, and request offsets. Traces are used to analyze workload behavior and can also be used to replay a workload's behavior for experimentation.

Definition 5. (Stage)

A request traverses multiple *stages* to access data. It first traverses a network stage to get to the server. It then traverses a storage stage to access the data on the storage device. It lastly traverses a network stage to return to the application with a response.

Definition 6. (End-to-end latency)

The *end-to-end latency* is the total time it takes a request to complete (i.e., completion time minus arrival time), including all of the time in the network and storage and all of the queueing.

Definition 7. (Tail latency Service Level Objective (SLO))

A *tail latency SLO* specifies a latency performance goal, which is described by an SLO latency (e.g., 150ms) and an SLO percentile (e.g., 99%). For example, we write workload W_1's SLO in Figure 1.1 as $99\% < 150ms$, which represents an SLO where 99% of W_1's requests have end-to-end latencies under 150ms. Different workloads are allowed to specify different SLOs, e.g., $W_2 : 99.9\% < 200ms$ and $W_3 : 90\% < 200ms$ and $W_4 : 99\% < 400ms$. These SLOs are each specified over a pair of workload and storage volume, which is known in literature as the pipe model. Applications accessing multiple storage volumes would specify SLOs for each storage volume.

1.3 Problem scope

Providing tail latency SLO guarantees is a broad problem with many open research questions. This section describes the scope of this thesis.

First, our work focuses on tail latency due to queueing, which is a major source of high tail latency. Improving the speed and/or tail latency of the underlying storage/network devices is complementary to our work. Instead, we take a black box approach where we assume the storage and network device performance can be profiled to extract information about tail latency, bandwidth, and throughput characteristics of devices. Small differences in device performance over time can be addressed by using a more conservative performance profile. To account for the performance peculiarities between different types of devices (e.g., SSD, disk), our profiling is specific to the type of device (see Section 2.3.2 for details).

Second, our work is designed for long-running user-facing workloads such as web servers or email servers. Our work is designed for open-loop workloads where requests are ultimately generated from external sources (i.e., users). We assume the expected user traffic load and burstiness can be upper bounded. It is impossible to guarantee tail latency SLOs with completely unpredictable workload behavior; an upper bound on traffic is necessary to reason about latency. In a sense, the upper bound on traffic specifies the amount of storage/network resource required to support the workload. Changes to workload behavior over time can be addressed by updating a workload's traffic requirements, but we assume this does not happen too frequently as it may trigger migrations.

Third, our work is designed for normal operating conditions. Mechanisms for handling failures are orthogonal to our work. For example, if storage is replicated for fault-tolerance, our work is useful for meeting SLOs for each replica. If storage in one replica fails, we cannot guarantee SLOs in that replica, but another replica will be able to respond within the desired tail latency SLO. The same applies to network failures that affect one replica. Network failures that significantly affect the network bandwidth of all replicas is outside the scope of our work. Also, incorporating our work into other fault-tolerance mechanisms is left to future work.

1.4 Goals

The goal of our work is to build a networked storage Quality of Service (QoS) system that can:

1. Meet each workload's end-to-end tail latency SLO
2. Efficiently utilize resources by admitting more workloads or minimizing the number of servers

Clearly, there is a tradeoff between achieving both goals. Using more resources will help meet SLOs due to reduced queueing. Using less resources will improve utilization, but may result in SLO violations due to high contention between workloads. Our work uses a variety of techniques including prioritization, rate limiting, admission control, and workload placement to strike the right balance between these goals.

1.5 Challenges

There are three primary challenges we face in sharing networked storage.

First, real-world workloads in production environments often exhibit bursty behavior. This burstiness often occurs at sub-second/second granularities, which does not significantly affect the average load, but has a large effect on tail latency. To meet tail latency SLOs, it is important to understand the interaction between the burstiness of multiple workloads sharing the system. Specifically, we need to have upper bounds on the queueing that could happen when workloads have bursts simultaneously.

Second, workloads are different both in their behavior and requirements, and they need to be treated accordingly. Specifically, workloads are bursty in different ways. One workload could have large, infrequent bursts. Another workload may have smaller, more frequent bursts. Yet another workload may have a medium, extended duration burst. Each of these behaviors affects the congestion within the system in different ways. Furthermore, each of the workloads may specify a different tail latency SLO. Our system needs to understand these differences between workloads to properly manage them.

Third, our goal is to build a unified QoS framework for both storage and networks, so we need to deal with the peculiarities of each type of resource. For storage, Solid State Drives (SSDs) and disks have their individual performance peculiarities (e.g., SSD writes are slower than reads whereas disks suffer from random access seek times). For networks, there are different workloads congesting at each network link. Furthermore, it is challenging to account for the end-to-end latency that spans both storage and network.

1.6 Thesis statement

Tail latency QoS systems should be designed to use a queueing analysis to properly quantify the effect of each workload's burstiness on tail latency. By incorporating a queueing analysis, a QoS system is able to pack workloads onto servers and determine a priori whether workloads can share a server while guaranteeing tail latency SLOs, which is not possible using reactive approaches.

Our new techniques demonstrate how to co-locate workloads to achieve better resource utilization than state-of-the-art approaches without violating SLOs. In particular, we study how to efficiently meet tail latency SLOs from the perspective of scheduling policies, admission control, and workload placement. We target real-world workloads, which are bursty both in their inter-arrival times and in their request sizes, running on multiple types of storage devices including magnetic disks and SSDs. Within this context, we answer the following questions:

- **Q1 (traffic enforcement):** How do we limit the impact of one workload on another?
- **Q2 (scheduling policy):** How should we arbitrate between shared workloads so that each workload can meet its SLO? How do we handle workloads with different SLO latencies (e.g., 400ms, 800ms, best-effort)?
- **Q3 (admission control):** How can we efficiently decide upfront whether to admit a new workload into the system while still guaranteeing all existing SLOs and the new workload's SLO?
- **Q4 (SLO percentiles):** How do we handle workloads with various SLO percentiles (e.g., 90th, 99th, 99.9th percentiles)?
- **Q5 (SLO-aware workload placement):** When dealing with thousands of workloads, which workloads should be co-located to best meet their SLOs while minimizing the resources used?

1.7 Prior work summary

Our research tackles questions spanning multiple problem domains, each with its set of related work. Despite all these existing works, our questions are still open. In this section, we highlight the most relevant work (see each chapter's related work for details).

Storage scheduling: Much of the prior work on storage scheduling is limited to the easier problem of sharing storage bandwidth [36, 38, 45, 67, 75, 79]. Sharing bandwidth is easier than sharing to meet latency goals because bandwidth is an average over time that is not affected by transient queueing. Some prior works target latency, but most of these works focus on the average latency [35, 46, 55, 56]. Looking at the average can mask some of the worst-case behaviors that often lead to stragglers.

A couple recent works, Cake [76] and Avatar [85], have considered tail latency SLOs at the 99th and 95th percentiles. Cake works by using reactive feedback-control based techniques. However, reactive approaches such as Cake do not work well for bursty workloads because bursts can cause a lot of SLO violations before one can react to them. Avatar [85] is an Earliest Deadline First (EDF) scheduler with rate limiting support, aimed at meeting the 95th percentile. Avatar suffers from three limitations. First, it does not address how to set rate limits. Second, its rate limiting model is not configurable for workloads of varying burstiness. Third, EDF scheduling does not generalize to networks since EDF relies on having a single entity that can timestamp and order requests. In our PriorityMeister work (Chapter 3), we show how to schedule multiple workloads to meet end-to-end tail latency SLOs in both storage and network, overcoming the limitations in Cake and Avatar.

Admission control: Recently, approaches for guaranteeing tail latency SLOs in datacenter networks [34, 43] have emerged. These works are based on a worst-case analysis known as Deterministic Network Calculus (DNC). Since DNC is a worst-case analysis, it is designed for analyzing the 100th percentile latency. Yet, DNC is still used despite the fact that typical users are only asking for SLOs at the 99th [81] or 99.9th [21] percentiles. This is done for two reasons. First, the mathematics for a worst-case analysis is far easier to understand than the mathematics for a tail percentile. Second, meeting a 100th percentile SLO implies meeting tail latency SLOs at lower percentiles (e.g., 99th, 99.9th). However, admission decisions based on the 100th percentile can be very conservative. The worst-case DNC analysis assumes all workloads behave adversarially where all workloads have bursts at exactly the same time, which is unrealistic in many settings.

The conservative nature of DNC-based admission control is known, and a new branch of theory called Stochastic Network Calculus (SNC) has been developed to address the shortcomings of DNC [13, 14, 17, 19, 25, 28, 48, 63, 64, 69, 82]. However, all of these works are only in theory and have never been applied in practice to computer systems. In our SNC-Meister work (Chapter 4), we show how to practically apply SNC for deciding admission in our networked storage system.

Workload placement: Most of the prior work on storage workload placement focus on load balancing [6, 8, 23, 37, 39, 61, 68]. While load balancing works well for providing fairness or throughput SLOs, it is not sufficient for tail latency SLOs. Tail latency is not only affected by the load of each workload, but also the burstiness. Little is known about how to pack workloads together while meeting tail latency SLOs. In our WorkloadCompactor work (Chapter 5), we show the importance of dynamically setting rate limits for workloads in order to co-locate more workloads while meeting SLOs.

1.8 Outline

In this section, we outline the key ideas and results for answering each of our questions, with details in the remainder of this thesis.

System architecture (Q1):

Chapter 2 describes our system architecture. We answer the first question, Q1, by describing the mechanisms for enforcing a workload's impact on another workload. Specifically, we explain how our system enforces priorities and rate limits for both storage and network traffic. For storage, we build a thin, transparent shim layer on top of Network File System (NFS) where we queue, prioritize, and rate limit NFS RPC requests. By using a shim layer, we are able to implement everything in userspace without any kernel modifications. For networks, we enforce priorities and rate limits via the Linux Traffic Control (TC) interface at each end-host. Network prioritization at the network switches is enforced via the Differentiated Services Code Point (DSCP) field (a.k.a. TOS IP field), where priorities are marked in this field using Linux TC at the end-hosts.

PriorityMeister (Q2):

Chapter 3 describes our PriorityMeister [86] work. We answer the second question, Q2, by comparing multiple scheduling policies and demonstrating how to configure priorities and rate limits to meet tail latency SLOs. We use prioritization to provide better latency for the workloads that need it most (i.e., low latency SLO). To prevent high priority workloads from starving low priority workloads, we use rate limiting to limit the impact of each of the workloads.

PriorityMeister is novel in that it analyzes the burstiness of workloads and uses queueing models from Deterministic Network Calculus (DNC) [50] as a building block for automatically selecting storage and network priorities and rate limits. PriorityMeister introduces an algorithm for finding a priority ordering that meets each workload's tail latency SLO. We also introduce the idea of using multiple rate limiters for a given workload to better characterize and limit a workload's burstiness. Our results show that PriorityMeister can meet tail latency SLOs with bursty workloads whereas reactive policies do not cope well with the burstiness found in real workloads.

SNC-Meister (Q3 & Q4):

Chapter 4 describes our SNC-Meister [87] work. We answer the third and fourth questions, Q3 and Q4, on how to provide admission control for SLOs at various tail percentiles (e.g., 99th, 99.9th). Specifically, we evaluate multiple admission control policies along two dimensions: how well they meet tail latency SLOs and how many workloads they admit. We find that state-of-the-art admission control systems can meet tail latency SLOs, but are conservative in the number of workloads they admit.

SNC-Meister also meets tail latency SLOs, but is able to admit many more workloads because it is based on a very new probabilistic theory called Stochastic Network Calculus (SNC) [27]. In fact, SNC-Meister is the first computer system to bring SNC to practice. SNC is designed to calculate tail latencies at any percentile, whereas all the state-of-the-art systems use the worst-case DNC theory, which is designed for the 100th percentile. Our results show that SNC-Meister meets workload SLOs while admitting 75% more workloads than DNC-based approaches targeting the 100th percentile.

WorkloadCompactor (Q5):

Chapter 5 describes our WorkloadCompactor work. We answer the fifth question, Q5, on how to best place workloads onto storage servers to minimize cost while meeting SLOs. Most prior works treat the placement problem as a load balancing problem [6, 8, 23, 37, 39, 61, 68]. However, with tail latency SLOs, both the load and burstiness of a workload affects the ability to co-locate workloads. A common way of representing load and burstiness is through the rate (r) and bucket size (b) parameters in token bucket rate limiters, where the rate represents the load and the bucket size represents the burstiness.

A key finding in our work is that the selection of the $\langle r, b \rangle$ parameters makes a big difference in the ability to pack workloads onto a server. Unfortunately, little is known on how to actually configure workload rate limits. WorkloadCompactor introduces a novel way of automatically selecting $\langle r, b \rangle$ parameters to minimize the number of servers needed to satisfy workload SLOs. WorkloadCompactor also introduces a scalable placement heuristic to quickly decide workload placements within seconds. Our results show that WorkloadCompactor uses 30-60% fewer servers than state-of-the-art approaches while meeting tail latency SLOs.

Chapter 2

System architecture

In this chapter, we describe the architecture of our new Quality of Service (QoS) system for tail latency Service Level Objectives (SLOs). Our QoS system adds the ability to provide tail latency guarantees to existing shared networked storage infrastructures within a datacenter. For example, it is suitable for cloud storage settings such as Amazon's Elastic Block Store (EBS) or OpenStack's Cinder. Our system can also be used in enterprise settings with storage servers running, for example, Network File System (NFS).

This chapter specifically addresses the question of what mechanisms do we use to limit the impact of one workload on another co-located workload. For example, our system needs to incorporate mechanisms that provide different levels of service to workloads based on their needs. We also need to include mechanisms for handling the burstiness that commonly arises in production workloads.

We first describe the overall design of our QoS system in Section 2.1. We then describe the two major components of our system: Section 2.2 describes our QoS enforcement, and Section 2.3 explains some key aspects of how we configure QoS parameters, with details in the following chapters.

2.1 System design

Our system originates from our IOFlow [71] work at Microsoft and is extended to work in Linux environments with NFS. Like IOFlow, our system takes a software-defined storage approach where the data plane is separated from the control plane.

Data plane: The data plane consists of components that handle the transport of data and enforce QoS policies. Section 2.2 describes the details in building our enforcement modules. We enforce priorities and rate limits at both the storage and network resources within our system to control the interference between workloads and provide better latency for the workloads that need it most (i.e., workloads having a low SLO latency and/or high SLO percentile). Using priority also allows our work to operate in existing systems alongside other best effort traffic. Any workloads that require a tail latency SLO would opt-in to our system to receive a higher priority, and all other best effort workloads would operate at the default lowest priority level. Lastly, to prevent starvation of low priority workloads, we enforce rate limits for each workload at the storage and network resources.

Control plane: The control plane in our system consists of a global controller that intelligently configures each of the enforcement modules with the appropriate configurations (e.g., priorities, rate limits). Section 2.3 describes the details in building our QoS configuration controller that analyzes workload traces to automatically configure QoS parameters. As input to our system, a user adds workload W by providing W's desired SLO latency (e.g., 150ms) and percentile (e.g., 99%) along with a representative trace of W's behavior. Traces contain historic access patterns as a list of requests parameterized by the arrival time, request type (e.g., read, write), request size (e.g., 4KB), and request offset. The trace needs to include traffic patterns that represent an upper bound on the expected behavior of the workload. Customers typically would capture traces over an extended period of time or during a high load period of the day. Alternatively, traces can be updated by running our system with a new trace. Robustness to deviations in trace behavior is evaluated in the subsequent chapters.

To understand the implications of selecting a representative trace, we first need to consider the differences between short-term burstiness and long-term load variations. Short-term burstiness denotes second/sub-second variations of a workload's bandwidth requirements. Long-term load variation denotes trends over the course of hours, such as diurnal patterns. While both types of variation affect latency, tail latency is mainly caused by transient queueing due to short-term burstiness. Short-term burstiness can lead to tail latency SLO violations even under low load: in our experiments, SLO violations occurred for utilizations as low as 40%. In production traces, we have seen short-term peaks with a rate that is 2 to 6 times higher than the average rate. By comparison, the difference between day-hour rates to night-hour rates is often less than a factor of 2. Our work focuses on capturing the effects of short-term burstiness on tail latency. Our experiments use real-world traces collected from from applications such as Microsoft Exchange, LiveMaps, and Ads servers running on Microsoft production servers [47].

2.2 QoS enforcement

Our system enforces two primary QoS mechanisms: priority and rate limits. We use strict prioritization to provide latency differentiation among the workloads (i.e., provide good latency to the workloads that require low latency). To prevent starvation, we rate limit each workload individually and only honor priority when workloads are within their rate limits.

Our rate limiters are built upon a leaky token bucket model that is parameterized by a rate *r* and a token bucket size *b*. When a request arrives, tokens are added to the token bucket based on the size of the request (see Section 2.3.2). If there is space in the bucket to add tokens without exceeding the configured token bucket size *b*, then the request is allowed to continue. Otherwise, the request is queued until there is sufficient space. Space becomes available as tokens continuously leak from the bucket at the configured rate *r*.

Since our system handles both storage and network resources, we have separate enforcement modules for each resource. Section 2.2.1 describes our storage enforcement, and Section 2.2.2 describes our network enforcement.

2.2.1 Storage enforcer

Our storage enforcer is responsible for scheduling requests at each of the storage devices. In our work, we implement storage prioritization and rate limiting on top of Network File System (NFS) running on commodity hardware. Since NFS is based on SunRPC, we hook into NFS at the Remote Procedure Call (RPC) layer without needing to resort to kernel modification. Our storage enforcement acts as a thin layer that intercepts and queues NFS RPC requests. Specifically, it creates queues for each workload and performs arbitration between the different workloads based on the priorities and rate limits assigned by our global controller (Section 2.3). Each workload's queue is serviced in first in first out (FIFO) order, and our storage enforcement layer executes requests from the highest priority (non-empty) queue where the workload is within its rate limits. Workloads exceeding its rate limit are treated as the default lowest priority level.

Our current storage enforcer implementation is designed to work with both Solid-State Drives (SSDs) and magnetic disks. Each type of storage device introduces unique challenges, in particular for enforcing priorities. The following sections describe the challenges with each type of storage and how our system addresses these challenges.

SSD challenges

Enforcing priorities in SSDs while maintaining good throughput is challenging. The most straightforward way to enforce priorities is to dispatch one request at a time to an SSD. However, dispatching requests one at a time does not work well for SSDs because modern SSDs require a high degree of parallelism to achieve high throughput[1].

[1]The reason behind this is that individual flash memory packages offer limited bandwidth which is commonly solved by bundling many packages together. In particular, modern SSDs employ parallelism at

While executing requests in parallel enables high throughput for SSDs, it also has the potential to interfere with the priority ordering. When a high priority request arrives at the storage system, it may need to wait for outstanding low priority requests. Also, SSDs may unintentionally delay a high priority request in order to more efficiently serve low priority requests. This can induce starvation for high priority requests while other requests are being served [83].

The reason behind these challenges is that SSDs are unaware of priority classes, and once a request has been dispatched to the SSD, we lose control over the request. Our current implementation addresses these issues from two angles. First, we limit the overall number of outstanding requests at the SSD as well as the overall number of bytes from outstanding requests. This allows us to exploit the SSD's parallel architecture while giving an upper bound on the time a newly-arriving high priority request needs to wait. Second, we limit the number of low priority requests as well as number of bytes that can be dispatched while a high priority request is in progress to prevent starvation of high priority requests.

Disk challenges

Enforcing priorities in disks is both easier and harder than in SSDs. On the one hand, disks do not require a high degree of parallelism to achieve good throughput; having one or two outstanding requests can achieve good utilization while maintaining good control over prioritization. On the other hand, disk performance is significantly impacted by sequential vs. random access behavior. If sequential write requests are executed one after another, then by the time the second request is issued, it will already be too late to write the next sequential chunk of data. The disk head will have already spun past the location to write the data, and an entire disk rotation is necessary. So our implementation specifically identifies sequential (or nearly sequential) writes and sends batches of these requests to the disk so they can be written in one pass. We also limit the number of requests in a batch to avoid clogging the disk with too many requests. Note that this only applies to sequential writes since sequential reads will find their data in the disk cache.

2.2.2 Network enforcer

Our network enforcer is responsible for prioritizing and rate limiting network traffic from each of the workloads. We build our network enforcer on top of the existing Linux Traffic Control (TC) infrastructure. The TC infrastructure allows users to build arbitrary QoS queueing structures for networking. On each end-host machine in our system, we use TC

many levels (e.g., channel-level, package-level, die-level etc.) [15, 22].

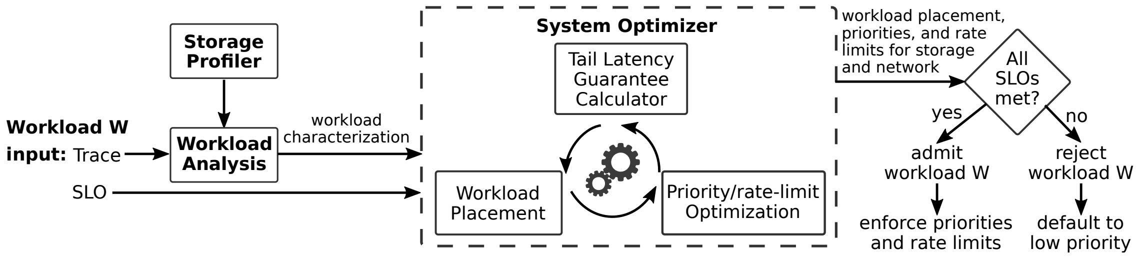

Storage Profiler
Workload W
input: Trace
SLO
Workload Analysis
workload characterization
System Optimizer
Tail Latency Guarantee Calculator
Workload Placement
Priority/rate-limit Optimization
workload placement, priorities, and rate limits for storage and network
All SLOs met?
yes
no
admit workload W
reject workload W
enforce priorities and rate limits
default to low priority

Figure 2.1: Flow chart of our QoS configuration controller: Configuring our system to meet tail latency SLOs when a new workload W arrives.

to configure PRIO queues for prioritization and Hierarchical Token Bucket (HTB) queues for rate limiting.

We first use a PRIO queue to separate packets into separate queues based on the workload's priority. We then use DSMARK to tag packet headers with priorities using the Differentiated Services Code Point (DSCP) flags (i.e., TOS IP field). The DSCP flags are used to prioritize packets within network switches whereas the PRIO queue is used to prioritize packets within the end-host machine. Our network switches support 7 levels of priority for each port, and using these priorities simply requires enabling the DSCP capability on the switch. Lastly, we use HTB queues for rate limiting each workload. To identify and route packets through the correct TC queues, we use filtering based on the source and destination IP addresses, which are different per VM.

2.3 QoS configuration controller

Figure 2.1 shows the process of adding a new workload, W, in our system. The user specifies as input to the system:

1. the SLO for workload W
2. a representative trace of W's behavior

The first step in our system is to analyze the workload's trace to characterize the workload's burstiness (see Section 2.3.1). As the performance of a workload depends on the underlying storage and network resources, the trace analysis step uses pre-computed performance profiles (see Section 2.3.2).

The second step in our system is to optimize the placement, priorities, and rate limits for the workloads across each of the storage and network stages (see Section 2.3.3). Our system is designed to handle the complexity of multiple stages, and it automatically optimizes QoS

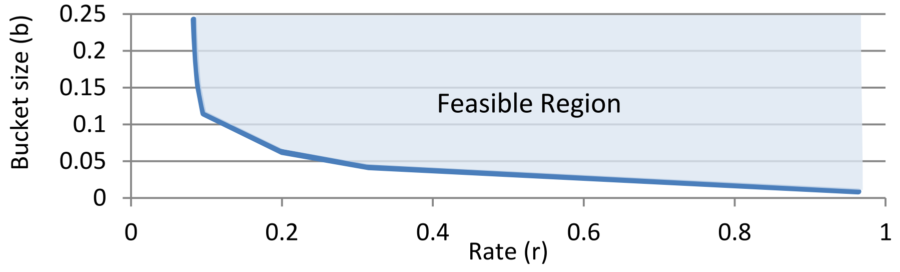


Figure 2.2: Characterizing burstiness via an *r*-*b* tradeoff curve. Feasible $\langle r, b \rangle$ points represent rate limit parameters such that a workload is not delayed by the rate limiter.

parameters for each stage while accounting for the end-to-end latency across stages. If our system finds a configuration that works, it admits the new workload and enforces the priorities and rate limits that were selected. Otherwise, the workload is rejected and would only be allowed to run in the default lowest priority level.

2.3.1 Workload analysis

One of the most important challenges we address in our work is characterizing the short-term burstiness that exists in many production workloads. Our investigation of storage traces [47] from multiple applications show that storage workloads are very bursty, and the bursts have varying durations and intensities. These bursts occur at the granularity of milliseconds to seconds. As a result, they don't significantly impact overall load, but have a large effect on tail latency.

The role of our trace analysis component is to analyze each workload's trace to build a mathematical model of the burstiness of the workload. Our research shows that one good way of characterizing burstiness is via a set of feasible rate limit $\langle r, b \rangle$ parameters. We define *feasible* $\langle r, b \rangle$ tuples for a workload as rate limit parameters such that the rate limiter is sufficiently large enough to service the workload's requests without delay. Figure 2.2 shows an example of feasible $\langle r, b \rangle$ tuples where all points on or above the *r*-*b* curve are feasible.

Our system provides a tool, `rbGen`, for generating *r*-*b* curves for storage and network based on a given trace. Algorithm 2.1 provides the pseudocode for `rbGen`. The tool sweeps across a given list of *r* values (e.g., 0.1, 0.2, ..., 1.0), and for each *r* value, it computes the minimum *b* such that the workload is not slowed down. These *b* values are computed by replaying the trace with infinite sized token buckets at each rate *r* and tracking the maximum tokens added at any point in time for each bucket. The output *b* values along with

Algorithm 2.1: *r-b* curve generation

```
// trace - list of requests in trace
// r - list of rates to sample in r-b curve
// tokensFunc - function to convert requests to tokens
// Returns: list of bucket sizes in r-b curve
// where <r[i], b[i]> are points on the r-b curve
// for i in [0, len(r))
function rbGen(var trace[], var r[], var tokensFunc)
{
  var b[len(r)]; // Initialized to 0
  var bucket[len(r)]; // Initialized to 0
  var prevTime = 0;
  for (req in trace) {
    var interarrival = req.arrivalTime - prevTime;
    for (var i = 0; i < len(r); i++) {
      // Drain token bucket for interarrival time
      bucket[i] -= r[i] * interarrival;
      if (bucket[i] < 0) {
        bucket[i] = 0;
      }
      // Add tokens for current request
      bucket[i] += tokensFunc(req);
      // Record max tokens added at any point
      if (bucket[i] > b[i]) {
        b[i] = bucket[i];
      }
    }
    prevTime = req.arrivalTime;
  }
  return b;
}
```

the input r values then form the $\langle r, b \rangle$ vertices in the piecewise linear r-b curve. To simplify the mathematics, all r-b curves are normalized (e.g., divide by network link bandwidth) such that $r = 1.0$ represents full bandwidth utilization.

It is important to note that the workload characterization depends on the storage or network stage. For example, network traffic into and out of the server are accounted for separately as two r-b curves since the amount of data transferred depends on the request type (e.g., read/write). Thus, a workload has an r-b curve for each of its stages: the network traffic to the server, the storage traffic at the server, and the network traffic leaving the server. `rbGen` accounts for the differences between stages via the `tokensFunc` argument, which converts request sizes to tokens of a given stage. The specifics of converting requests to tokens depend on the underlying resource type (e.g., SSD), and we next describe the process in Section 2.3.2.

2.3.2 Profiling

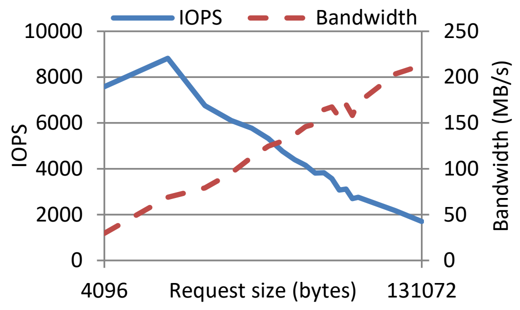
Storage and network resources each have unique properties that affect the performance of a request. To work with these different resources in a single analysis framework, we abstract the variety of request types (e.g., read, write) and request sizes into a single common metric, tokens. Our system incorporates storage and network models to determine the number of tokens associated with a request. To represent tokens in networks for example, the network traffic leaving the server would use the number of bytes accessed for read requests and a constant (i.e., size of acknowledgment) for write requests. For storage, we implement a storage model to represent the amount of "work" introduced by a request based on measured storage performance profiles.

Storage model

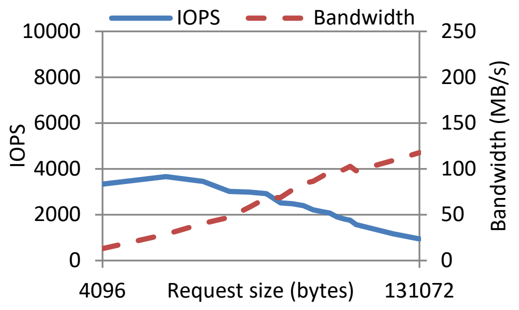
To model the performance of storage, our system includes a profiler that empirically measures the amount of "work" generated by a request. Work is measured in units of time and denotes the time consumed by a request without the effects of queueing. We use work as a representation of tokens in our token bucket rate limiters.

Our current implementation includes storage profilers for SSDs and disks. We expect our system to extend to other storage devices such as RAID arrays by adapting our profiler to the specific peculiarities of the device.

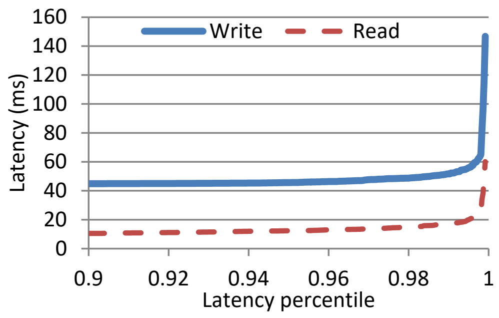
SSDs are complex devices with many performance peculiarities. SSD performance cannot be described with a single parameter, but rather requires profiling the device across various access types. In particular, read and write throughput is very different for SSDs. Writes may need to erase SSD blocks, which is considerably slower than reading SSD



(a) Read throughput (IOPS/Bandwidth)



(b) Write throughput (IOPS/Bandwidth)



(c) Tail latency

Figure 2.3: Performance profile of NFS storage stack running with our SSD.

blocks. To accurately profile a SSD, we profile reads and writes separately. Additionally, the request size significantly impacts SSD throughput. For small requests, SSDs are limited to the maximum IOPS supported by the device. For large requests, SSDs are limited to the maximum bandwidth supported by the device. Our system builds a performance profile (e.g., Figure 2.3(a) and Figure 2.3(b)) for each SSD by measuring the empirical throughput over a range of request sizes. The performance profile includes the performance of both the SSD and storage stack so as to have a holistic view of the storage subsystem.

These SSD profiles are used to compute the amount of "work" induced by a request. Our system uses this generic notion of work to quantify the congestion between workloads at an SSD. We calculate the work induced by a request by taking the inverse of the IOPS throughput (i.e., $work = \frac{1}{IOPS}$), where IOPS denotes the number of I/O operations per second.

In addition to the work generated by a request, there is also a tail latency effect due to the SSD and storage stack. For example, writes are sometimes delayed to allow more write batching. Thus, our system also profiles the tail latency of requests without the queueing effects of bursty workloads to isolate the SSD and storage stack tail latency (e.g., Figure 2.3(c)). This profiled latency is then added to the estimated queueing latency for a request. Our storage model is most similar to the table-based approach in [7] where we build tables for (i) throughput and (ii) a base time to service a request.

For disks, we also account for the request offset and distance between subsequent request offsets (i.e., offset – previous offset) to compensate for disk seek time. We keep a history of requests to identify sequential accesses that do not introduce a seek delay.

Network model

Translating network performance to tokens is much easier than in storage. We simply use bytes as a representation for tokens and account for the number of bytes transmitted by a request. The number of bytes transmitted not only depends on the request size, but also the request type. For read requests, there is a small request sent to the server and a large response back from the server with the data. For write requests, there is a large request sent to the server with the data and a small response back from the server.

2.3.3 Optimization

Once we have workload characterizations, the next step (see Figure 2.1) is to optimize the placement, priorities, and rate limits for the workloads. First, the placement component identifies candidate servers upon which to place the workload. Second, the optimizer component selects priorities and rate-limits for the workloads. Third, the latency checker component determines whether the configuration (i.e., placement, priorities, rate limits) would satisfy all workload SLOs. If all SLOs are satisfied, then our system admits the new workload and configures our enforcement modules to enforce the priorities and rate limits for each workload. If not, the cycle begins again with the placement component identifying a new candidate server. If there are no more servers, then the new workload is rejected and can only run in the default lowest priority level.

Optimizing the workload configuration is the major body of our work and is addressed in the following chapters. Workload placement is investigated in Chapter 5. Priority optimization is investigated in Chapter 3. Rate-limit optimization is investigated in Chapter 5. Checking latency guarantees is investigated in Chapter 3 and Chapter 4.

Chapter 3

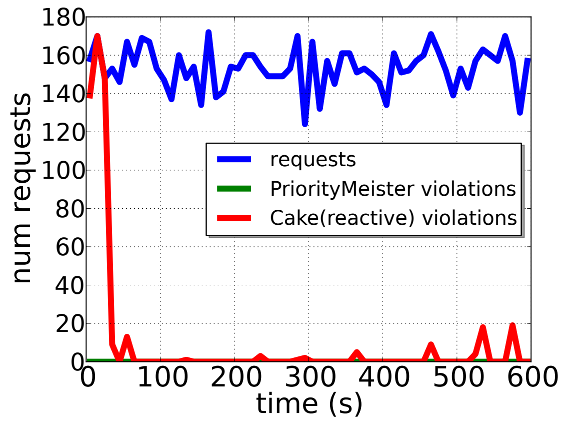
PriorityMeister: Tail latency QoS for shared networked storage

In this chapter, we explore the question of how to arbitrate between multiple workloads sharing a networked storage system. In particular, we compare multiple scheduling policies and evaluate how well they meet tail latency SLOs for each workload. A key challenge we face in meeting tail latency SLOs is dealing with the burstiness that occurs in many production workloads. While burstiness does not significantly affect the average system load, it has a large effect on tail latency.

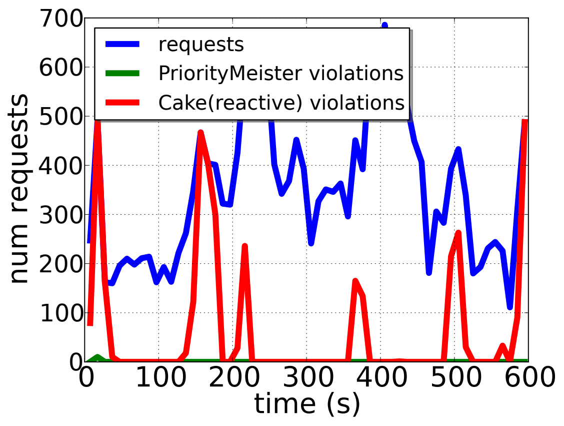
A common approach for meeting SLOs is to use a reactive feedback-control loop to give more or less of a resource to a workload based on how well it meets its SLOs. However, under bursty workloads as seen in practice, reactive policies do not work in meeting tail latency SLOs because they cannot react quickly to bursts. Exceeding the SLO even for small periods of time can lead to SLO violations at tail percentiles.

We present PriorityMeister, a new approach for meeting tail latency SLOs. PriorityMeister is different from prior approaches in that it uses a tail latency calculator that provides mathematical guarantees for meeting tail latency SLOs. PriorityMeister's tail latency calculator uses a branch of theory called Deterministic Network Calculus (DNC) to derive tight upper bounds on the worst-case latency of each workload. We demonstrate how to apply DNC to the end-to-end latency spanning both storage and network resources. We also show how PriorityMeister uses DNC to more intelligently configure a combination of per-workload priorities and rate limits to meet each workload's tail latency SLO. In real system experiments and under production trace workloads, PriorityMeister outperforms most recent reactive request scheduling approaches, with more workloads satisfying latency SLOs at higher latency percentiles.

We introduce the problem and discuss the scope of this chapter in Section 3.1. We



(a) synthetic trace – low burstiness



(b) real trace – high burstiness

Figure 3.1: Illustration of the effect of request burstiness on latency SLO violations. The two graphs show time series data for a real trace, (b), and a synthetic trace with less burstiness, (a), each co-located with a throughput-oriented batch workload. Each graph has three lines: the number of requests in a 10-second period (blue), the number of SLO violations using a state-of-the-art reactive approach (red), and the number of SLO violations using PriorityMeister (green). The reactive approach (Cake [76]) is acceptable when there is little burstiness, but incurs many violations when bursts occur. PriorityMeister (PM) provides more robust QoS behavior with almost 0 SLO violations. Details of the system setup, traces, algorithms, and configurations are described later in Section 3.3. These graphs are provided up front only to illustrate the context and contribution of the new approach.

present the design and implementation of PriorityMeister in Section 3.2. We then describe our experimental setup in Section 3.3 followed by our results in Section 3.4. We discuss related work in Section 3.5 and conclude with a summary of this chapter in Section 3.6.

3.1 Introduction

Meeting end-to-end tail latency service level objectives (SLOs) is challenging, particularly for bursty workloads found in production environments. First, tail latency is largely affected by queueing, and bursty workloads cause queueing for all workloads sharing the underlying infrastructure. Second, the end-to-end latency is affected by all the *stages* in a request (e.g., accessing storage, sending data over network), and queues may build up at different stages at different times.

Much of the prior work on storage scheduling is limited to the easier problem of sharing storage bandwidth [36, 38, 45, 67, 75, 79]. Sharing bandwidth is easier than latency because bandwidth is an average over time that is not affected by transient queueing. Some prior works target latency, but most of these works are focused on the average latency [35, 46, 55, 56]. Looking at the average can hide some of the worst-case behaviors that often lead to stragglers.

A recent work, Cake [76], has considered tail latency QoS at the 99th percentile. Cake works by using reactive feedback-control based techniques. However, reactive approaches such as Cake do not work well for bursty workloads because bursts can cause a lot of SLO violations before one can react to them. Figure 3.1 illustrates this point; analysis and more experiments appear later. Figure 3.1(a) shows that when the request rate (blue line) is not bursty, Cake meets latency SLOs with infrequent violations (red line). Figure 3.1(b), on the other hand, shows that when the workload is more bursty as in a real trace, Cake has periods of significant SLO violations. The difficulties in meeting latency SLOs are further compounded when dealing with multiple stages since the end-to-end latency is composed of the sum of all stage latencies.

This chapter introduces PriorityMeister (PM), a new proactive QoS system that achieves end-to-end tail latency SLOs across multiple stages through a combination of priority and token-bucket rate-limiting. PriorityMeister works by analyzing each workload's burstiness and load at each stage. This, in turn, is used to calculate per-workload token-bucket rate limits that bound the impact of one workload on the other workloads sharing the system. As we will see in Section 3.2.1, a key idea in PriorityMeister is to use *multiple* rate limiters simultaneously for each workload at each stage. Using multiple rate limiters simultaneously allows PriorityMeister to better bound the burstiness of a workload.

Rate limiting alone is insufficient because workloads have different latency requirements and different workload burstiness. Thus, workloads need to be treated differently to meet their latency SLOs and bound the impact on other workloads. PriorityMeister uses priority as the key mechanism for differentiating latency between workloads. Note that priority is used to avoid delaying requests from workloads with tight latency requirements rather than to prioritize workloads based on an external notion of importance. Manually setting priority is typical, but is laborious and error-prone. Indeed, simultaneously capturing the effect of each workload's burstiness on lower priority workloads is hard. PriorityMeister builds a model to estimate the worst-case per-workload latency based on the burstiness for each workload. Our model is based on *deterministic network calculus*, an analysis framework for worst-case queueing estimation. Using our analytical model, PriorityMeister quickly searches over a large space of priority orderings at each stage to automatically set priorities to meet SLOs.

PriorityMeister also supports different per-workload priorities and rate limits at each stage (as opposed to a single priority throughout). Rather than having one workload that is highest-priority throughout and a second that is lower priority throughout, where the first workload meets its SLO and the second doesn't, we can instead have both workloads be highest-priority at some stages and lower priority at others. Since a workload may not need the highest priority everywhere to meet its SLO, this mixed priority scheme can allow more workloads to meet their SLOs.

In this chapter, we make the following main contributions. First, we develop an algorithm for automatically determining the priority and rate limits for each of the workloads at each stage to meet end-to-end tail latency SLOs. PriorityMeister achieves these goals by combining deterministic network calculus with the idea of using multiple rate limiters simultaneously for a given workload. Second, we build a real QoS system consisting of network and storage where we demonstrate that PriorityMeister outperforms state of the art approaches like Cake [76]. We also compare against a wide range of other approaches for meeting SLOs and show that PriorityMeister is better able to meet tail latency SLOs (see Figure 3.3, Figure 3.4, and Figure 3.9), even when the bottleneck is at the network rather than storage (see Figure 3.7). Third, we show that PriorityMeister is robust to mis-estimation in storage performance (see Figure 3.8), varying degrees of workload burstiness (see Figure 3.5), and workload misbehavior (see Figure 3.6). Fourth, we develop a *simple* heuristic, which we call *bySLO*, and we show that it performs surprisingly well, also outperforming Cake (see Figure 3.9).

3.2 PriorityMeister

PriorityMeister provides QoS on a per-workload basis. Each workload runs on a client VM and accesses storage at a server VM. An application with multiple client or server VMs can be represented as multiple workloads with the same SLO.

Workloads consist of a stream of requests from a client to a server and back, where each request is characterized by an arrival time, request type (e.g., read, write), request size, and request offset. A request comprises three stages: the network request from the client to server, the storage access at the server, and the network reply from the server back to the client. For each of the network stages, there are queues at each machine and network switch egress port. For the storage stage, there is a queue at each storage device. Each stage has independent priorities and rate limits for each workload, which are determined by PriorityMeister. Details on our system architecture are described in Chapter 2.

The two primary QoS parameters that PriorityMeister automatically configures are priority and rate limits. Prioritization is our key mechanism for providing latency differentiation among the workloads. We use strict priority to provide good latency to the workloads that require low latency. To prevent starvation, we use rate limiting and only honor priority when workloads are within their rate limits. PriorityMeister is unique in that we deploy *multiple* rate limiters for each workload at each stage.

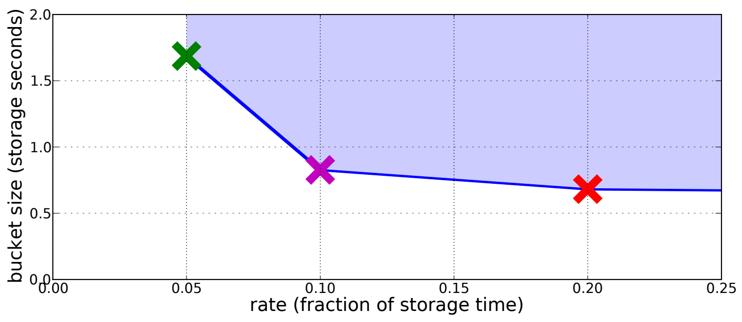
PriorityMeister's automatic QoS configuration provides an interesting alternative to having system administrators manually setting priorities and rate limits, which is both costly and error prone. Users are inherently bad at choosing QoS parameters since they may not be aware of the other workloads in the system. PriorityMeister chooses priorities and rate limits automatically using high-level SLOs, which are much easier for a user to specify. Section 3.2.1 describes how PriorityMeister sets rate limits for each workload. Section 3.2.2 describes how PriorityMeister prioritizes workloads based on our latency analysis model, described in Section 3.2.3.

3.2.1 Setting rate limits

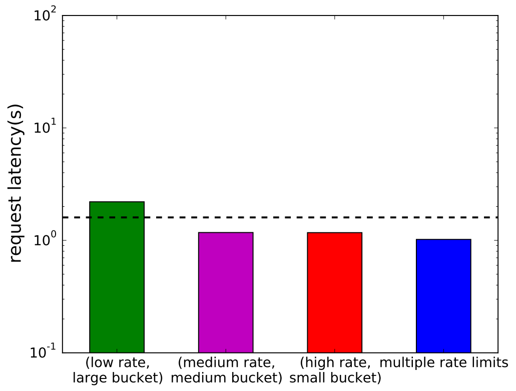
Our rate limiters are based on a leaky token bucket model that is parameterized by a rate r and a token bucket size b. When a request arrives, tokens are added to the token bucket based on the request size. If there is space in the bucket to add tokens without overflowing the bucket, then the request is allowed to continue. Otherwise, the request is queued and waits until enough tokens drain out of the bucket at the constant rate r. The rate corresponds to the bandwidth consumed by the workload, and the token bucket size corresponds to the burstiness of the workload.

One of the key contributions in PriorityMeister is a novel way of rate limiting using *multiple* token bucket rate limiters simultaneously for the same workload. This allows the system to more accurately limit the effect of one workload on another. We show an example motivating this idea in Figure 3.2, which is described in detail in the next paragraph. The notion of using multiple rate limiters simultaneously for a workload is unusual and is *not* the same as using the minimum rate and token bucket size. Using multiple rate limiters means that when a request arrives, the same number of tokens are added to each of the multiple token buckets. If there is space in *all* of the token buckets to add tokens without overflowing each bucket, then the request is allowed to continue. But if *any* of the buckets does not have enough space, then the request must wait for tokens to drain out of the buckets at their corresponding rates until there is space in all of the buckets.

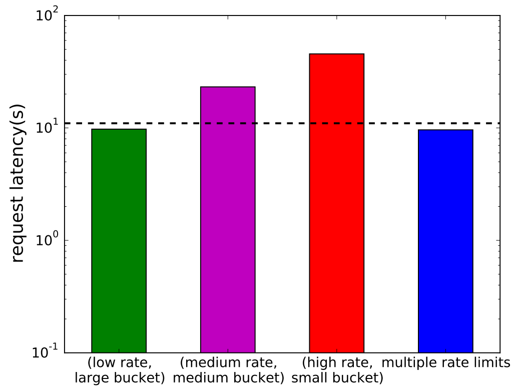
Figure 3.2 shows an example motivating the idea of multiple rate limiters on a high priority workload W_H. Note that there are many rate limit parameters $\langle r, b \rangle$ that are sufficiently high to allow the workload to proceed without any queueing. Figure 3.2(a) shows an example of the rate limit parameters (Workload B in Table 3.1) where all of the points in the shaded region allows workload W_H to proceed without queueing. *We use this shaded region as a characterization of workload W_H's behavior.* Section 2.3.1 describes the process of calculating this region.



(a) rate limit pairs of high priority workload



(b) 99.9% latency of medium priority workload



(c) 99.9% latency of low priority workload

Figure 3.2: How token bucket parameters $\langle \text{rate}, \text{bucket size} \rangle$ of a high priority workload affects lower priority workloads. In this experiment, we run 3 co-located workloads at different priorities. Figure 3.2(a) shows the set of rate limit parameters for the high priority workload where all of the points in the shaded region allows the workload to proceed without queueing. We then cause the high priority workload to misbehave by inducing a large 40 second burst of work in the middle of the trace. Figure 3.2(b) and Figure 3.2(c) show the effect, respectively, on the medium priority and low priority workload when the high priority workload misbehaves. Each colored X in Figure 3.2(a) corresponds to a similarly colored bar in Figure 3.2(b) and Figure 3.2(c), representing the 99.9th percentile latency of the medium and low priority workloads when the high priority workload is rate limited by the parameters marked by the X. When using only one rate limiter (one X) on the high priority workload, the lower priority workloads are not able to meet their SLOs (dashed line). PriorityMeister uses multiple rate limiters simultaneously (blue bar) to allow both lower priority workloads to meet their SLOs.

Now to investigate how the choice of W_H's rate limit affects performance, we try each of the $\langle r, b \rangle$ rate limits marked by colored X's and show the effect on lower priority workloads. Figure 3.2(b) shows the 99.9th percentile latency of a medium priority workload W_M, and Figure 3.2(c) shows the 99.9th percentile latency of a low priority workload W_L. If we select the $\langle \text{low rate}, \text{large bucket} \rangle$ rate limit for the high priority workload W_H (green X in Figure 3.2(a)), then the medium priority workload W_M exceeds its SLO since it is delayed by large bursts of workload W_H (green bar in Figure 3.2(b) is above horizontal dashed SLO line). If we select the $\langle \text{medium rate}, \text{medium bucket} \rangle$ or $\langle \text{high rate}, \text{small bucket} \rangle$ rate limit for W_H, then the low priority workload W_L exceeds its SLO (see Figure 3.2(c)) since there is insufficient bandwidth leftover once W_H consumes a medium or high rate. Thus, none of the rate limits individually allows the system to meet SLOs for both W_M and W_L. So in PriorityMeister, we instead select multiple rate limits (all 3 X's) simultaneously for W_H, which allows both W_M and W_L to meet their SLOs (blue bar). *Using multiple rate limits simultaneously allows PriorityMeister to more accurately characterize and constrain W_H without delaying W_H. This in turn helps W_M and W_L meet their SLOs.*

3.2.2 Setting priorities

PriorityMeister introduces a new prioritizer algorithm that efficiently finds a priority ordering that can meet tail latency SLOs. That is, we want to determine priorities for each stage of each workload such that the workload's worst-case latency, as calculated by the latency analysis model (Section 3.2.3), is less than the workload's SLO. While the size of the search space appears combinatorial in the number of workloads, we have a key insight that makes the search polynomial: *if a workload can meet its SLO with a given low priority, then the particular ordering of the higher priority workloads does not matter*. Only the cumulative effects of higher priority workloads matter. Thus, our algorithm tries to assign the lowest priority to each workload, and any workload that can meet its SLO with the lowest priority is assigned that priority and removed from the search. Our algorithm then iterates on the remaining workloads at the next lowest priority.

If we come to a point where none of the remaining workloads can meet their SLOs at the lowest priority, then we take advantage of assigning a workload *different* priorities at each stage (e.g., setting a workload to have high priority for storage but medium priority for network, or vice versa). Specifically, consider the remaining set of workloads that have not yet been assigned priorities. For each workload, w, in this set, we calculate w's *violation*, which is defined to be the latency estimate of w minus the SLO of w, in the case that w is given lowest priority in the set across all stages. For that workload, w, with smallest violation, we determine w's worst-case latency at each stage. For the stage where w has smallest latency, we assign w to be the lowest priority of the remaining set of workloads.

We then repeat the process until all workloads have been assigned priorities at each stage.

3.2.3 Calculating latency estimates

PriorityMeister incorporates a latency analysis model to estimate worst-case latencies for the workloads under a given priority ordering and rate limits. The model we use in our system is based on the theory of deterministic network calculus. The main concepts in deterministic network calculus are arrival curves and service curves. An *arrival curve* $\alpha(t)$ is a function that defines the maximum number of bytes that will arrive in any period of time t. A *service curve* $\beta(t)$ is a function that defines the minimum number of bytes that is guaranteed to be serviced in any period of time t. For clarity in exposition, we describe the latency analysis model in terms of bytes, but our solution works more generally in terms of tokens, which are bytes for networks and storage "work" for storage (see Section 2.3.2). Deterministic network calculus proves that the maximum horizontal distance between a workload's arrival curve and service curve is a tight worst-case bound on latency. Thus, our goal is to calculate accurate arrival and service curves for each workload.

In our rate-limited system, an arrival curve $\alpha_w(t)$ for workload w is formally defined $\alpha_w(t) = \min\limits_{i=1,\ldots,m} (r_i * t + b_i)$ where workload w has m rate limit pairs $\langle r_1, b_1\rangle$, ..., $\langle r_m, b_m\rangle$. The challenge is calculating an accurate service curve, and we resort to using a linear program (LP) for each workload w. Our approach is similar to the technique used in [11], which has been proven to be correct. To calculate the service curve $\beta_w(t)$ for workload w, we build a worst-case scenario for workload w by maximizing the interference on workload w from higher priority workloads. Instead of directly calculating $\beta_w(t)$, it is easier to think of the LP for the inverse function $\beta_w^{-1}(y)$. That is, $t = \beta_w^{-1}(y)$ represents the maximum amount of time t that it takes workload w to have y bytes serviced.

We use the following set of variables in our LP: t_{in}^q, t_{out}^q, R_k^q, $R_k'^q$. For each queue q, t_{in}^q represents the start time of the most recent backlog period before time t_{out}^q. That is, queue q is *backlogged* (i.e., has work to do) during the time period $[t_{in}^q, t_{out}^q]$. Note that queue q may be backlogged after t_{out}^q, but not at time t_{in}^q. R_k^q represents the cumulative number of bytes that have arrived at queue q from workload k at time t_{in}^q. $R_k'^q$ represents the cumulative number of bytes that have been serviced at queue q from workload k at time t_{out}^q. Throughout, k will represent a workload of higher priority than w.

The constraints in our LP are as follows:

Time constraints: For each queue q, we add the constraint $t_{in}^q \leq t_{out}^q$ to ensure time is moving forward. For all queues q and for all queues q' that feed into q, we add the constraint $t_{out}^{q'} = t_{in}^q$ to relate times between queues.

Flow constraints: For each queue q and for each workload k in queue q, we add the constraint $R_k^q \leq R_k'^q$. Since the queue is empty, by construction, at the start of the backlog period (t_{in}^q), all the bytes that have arrived (R_k^q) by time t_{in}^q must have been serviced. Consequently, this constraint ensures that the cumulative number of bytes serviced is non-decreasing over time.

<u>Rate limit constraints</u>: We need to constrain the extent to which other workloads, k, can interfere with workload w. For a particular workload k, let $\langle r_1, b_1 \rangle$, ..., $\langle r_m, b_m \rangle$ be its rate limit parameters, and let q^* be workload k's first queue. Then for each queue q containing workload k, we add the constraints $R_k'^q - R_k^{q^*} \leq r_i * (t_{out}^q - t_{in}^{q^*}) + b_i$ for each rate limit pair $\langle r_i, b_i \rangle$. These constraints apply rate limits to each of the relevant time periods for workload k, and are added for each workload k.

<u>Work conservation constraints</u>: For each queue q, we need to ensure that bytes are being serviced when there is a backlog. Let B_q be queue q's bandwidth. Since each queue q is backlogged during time period $[t_{in}^q, t_{out}^q]$ by construction, the queue must be servicing requests at full bandwidth speed between t_{in}^q and t_{out}^q, which yields the constraint $\sum_k (R_k'^q - R_k^q) = B_q * (t_{out}^q - t_{in}^q)$ where we sum over the workloads k in queue q, including w.

<u>Objective function</u>: The LP's goal is to maximize the amount of time needed for workload w to have y bytes serviced. Let q_1 and q_n be the first and last queues of workload w respectively. We add the constraint $R_w'^{q_n} - R_w^{q_1} = y$ to ensure that y bytes are serviced. Then, our objective function is to maximize $t_{out}^{q_n} - t_{in}^{q_1}$.

3.3 Experimental setup

In our experiments, a workload corresponds to a single client VM that makes requests to a remote NFS-mounted filesystem. Each workload has a corresponding trace file containing its requests. The goal of each *experiment* is to investigate the tail latency when *multiple* workloads are sharing storage and network, so each of our experiments use a mixture of workloads.

3.3.1 Comparison approaches

In our experiments, we compare 5 QoS approaches: Proportional fair-share (ps), Cake [76], Earliest Deadline First (EDF), prioritization in order by SLO (bySLO), and PriorityMeister (PM).

Proportional sharing (ps)

We use proportional sharing as a strawman example of a system without latency QoS, where each workload gets an equally weighted share of storage time, and no network QoS is used.

We do not expect ps to be good at meeting latency SLOs.

Cake [76]

We implement the algorithm found in the recent Cake paper as an example of a reactive feedback-control algorithm. Cake works by dynamically adjusting proportional shares to meet latency SLOs. We use the same control parameters as found in the paper except for the upper bound SLO-compliance parameter, which we increase to improve the tail latency performance. To avoid any convergence issues, we only measure performance for the second half of the trace in all of our experiments. Since the Cake paper only supports a single latency sensitive workload, we extend the Cake algorithm to support multiple latency sensitive workloads by assigning a weight to each workload, which is adjusted using the Cake algorithm.

We have also tried extending Cake to support network QoS. Since networks do not have an easy way of dynamically updating proportional shares, we use rate limits as a proxy for proportional shares. We assign a weight to each workload as before and use a DRF-like [32] algorithm to assign rate limits based on the weights. Our initial experiments indicate that this rate limiting hurts more than it helps, so our results in Section 3.4 drop this extension, and we do not use network QoS with a Cake model.

Earliest Deadline First (EDF)

We implement an EDF policy in our storage enforcer, and we configure the deadlines for each workload as the workload's SLO. There is no straightforward way of extending an EDF policy to networks, so we do not use network QoS with this policy.

Prioritization by SLO (bySLO)

We also investigate a simple policy, that we have not seen in prior literature, where we assign workload priorities in order of the workload latency SLOs. That is, we assign the highest priority to the workload with the tightest SLO. This is supported for both network and storage.

PriorityMeister (PM)

PriorityMeister is our primary policy that we compare against the other policies and is described in Section 3.2.

Workload label	Workload source	Estimated storage load	Estimated network load	Interarrival Variability, C_A^2
Workload A	DisplayAds trace	5%	5%	1.3
Workload B	MSN storage trace	5%	5%	14
Workload C	LiveMaps trace	55%	5%	2.2
Workload D	Exchange trace (behaved)	10%	5%	23
Workload E	Exchange trace (misbehaved)	$>100\%$	15%	145
Workload F	Low burst trace	25%	5%	1
Workload G	High burst trace	25%	5%	20
Workload H	Very high burst trace	25%	5%	40
Workload I	Medium network load trace 1	35%	20%	1
Workload J	Medium network load trace 2	45%	25%	1
Workload K	Ramdisk trace	N/A	35%	3.6
Workload L	Large file copy (throughput)	N/A	N/A	N/A

Table 3.1: Workload traces used in evaluating PriorityMeister. Workloads A-E are from production servers [47] and workloads F-L are synthetic.

3.3.2 Traces

We evaluate our system implementation using a collection of real production storage traces (described in [47]) and synthetic traces. Each trace contains a list of requests parameterized by the arrival time, request size, request type (e.g., read, write), and request offset. Table 3.1 provides a description of the traces used in our evaluation. We show the estimated load on the storage and network, as well as the squared coefficient of variation of the inter-arrival times (C_A^2), which gives one notion of burstiness. For the synthetic traces, a C_A^2 of 1 indicates a Poisson arrival process, and higher values indicate more bursty arrival patterns.

As discussed in [66], there are vast differences when replaying traces in an open loop vs. closed loop fashion. To properly represent end-to-end latency and the effects of queueing at the client, we replay traces in an open loop fashion. Closed loop trace replay masks a lot of the high tail latencies since much of the queueing is hidden from the system. Closed loop trace replay is designed for throughput experiments, and we use this form of replay solely for our throughput-oriented workload (Workload L).

3.3.3 SLOs

Each workload in a given experiment has its own latency SLO, which is shown in the results section as a horizontal dashed line. The SLO represents the maximum end-to-end latency that a workload considers acceptable. The end-to-end latency is defined as the difference between the request completion time and the arrival time from the trace, which includes all of the queueing time experienced by the requests.

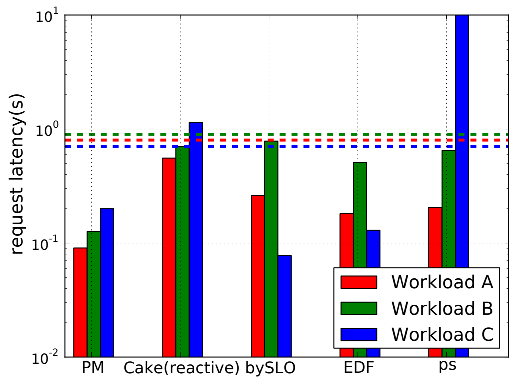
Not all requests in a workload will necessarily meet its SLO, so we also use the metric of a latency percentile to measure how many requests failed its SLO. For example, meeting the SLO for the 99th percentile means that at least 99% of the workload's requests had a latency under the desired SLO.

3.3.4 Experimental testbed

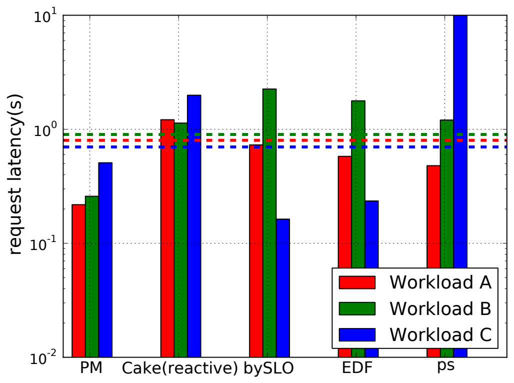
All experimental results are collected on a dedicated rack of servers. The client and storage nodes are all Dell PowerEdge 710 machines, each configured with two Intel Xeon E5520 processors, 16GB of DRAM, and 6 1TB 7200RPM SATA disk drives. Ubuntu 12.04 with 64-bit Linux kernel 3.2.0-22-generic is used for the host OS, and virtualization support is provided by the standard kvm package (qemu-kvm-1.0). Ubuntu 13.10 with 64-bit Linux kernel 3.11.0-12-generic is used as the guest operating system. We use the standard NFS server and client that comes with these operating systems to provide remote storage access. The top-of-rack switch is a Dell PowerConnect 6248 switch, providing 48 1Gbps ports and 2 10Gbps uplinks, with firmware version 3.3.9.1 and DSCP support for 7 levels of priority.

3.4 Results

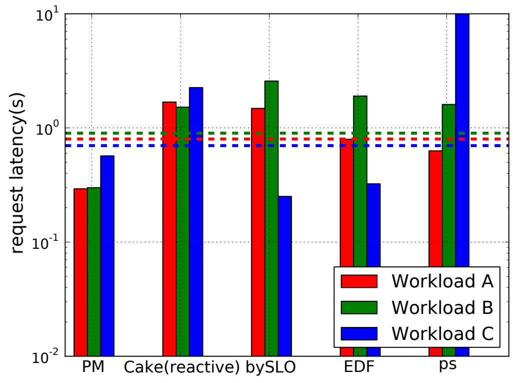
This section evaluates PriorityMeister (PM) in comparison to other state of the art policies across multiple dimensions. Section 3.4.1 demonstrates the ability of PriorityMeister in meeting tail latency SLOs on a set of production workload traces [47]. We find that PriorityMeister is able to take advantage of its knowledge of workload behaviors to meet all the SLOs whereas the other policies start to miss some SLOs above the 99th percentile tail latency. Section 3.4.2 investigates the differences between proactive (PriorityMeister) and reactive (Cake [76]) approaches, as we vary the burstiness of a workload. As burstiness increases, reactive approaches have a harder time adapting to the workload behavior and meeting SLOs. PriorityMeister is able to quantify the burstiness of workloads and safely prioritize workloads to meet SLOs. Section 3.4.3 then proceeds to show that PriorityMeister's prioritization techniques are safe to workload misbehavior through its automatic configuration of rate limits.



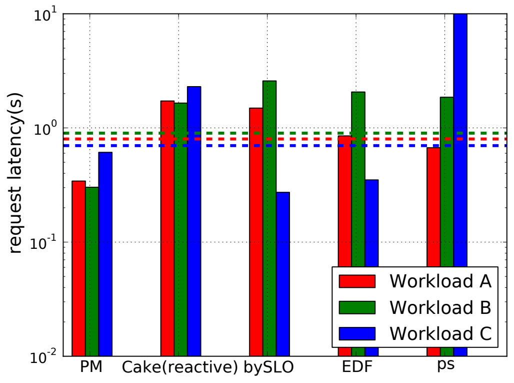
(a) 90th percentile



(b) 99th percentile



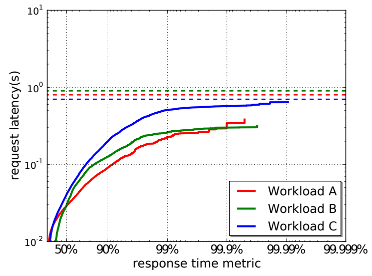
(c) 99.9th percentile



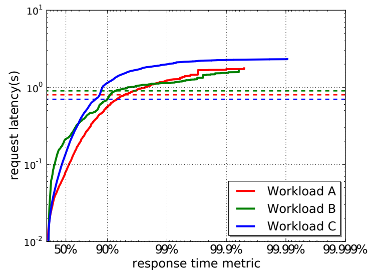
(d) 99.99th percentile

Figure 3.3: Comparing PriorityMeister to other scheduling policies. PriorityMeister (PM) is the only policy that satisfies all SLOs across all percentiles. In this experiment, we replay three latency-sensitive workloads derived from production traces (Workloads A, B, and C, from Table 3.1) sharing a disk with with a throughput-oriented workload (Workload L; not shown) that represents a file copy. Each of the colored horizontal dashed lines correspond to the latency SLO of the similarly colored workload. Each subgraph shows a different request latency percentile. Each group of bars shows the latency of the three workloads under each scheduling policy (described in Section 3.3.1).

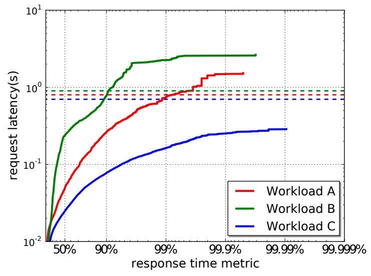
In Section 3.4.4, we investigate scenarios when the bottleneck shifts from storage to network. We show that PriorityMeister's techniques continue to work when the network becomes a bottleneck, whereas the other state of the art policies do not generalize to networks. We conclude with a sensitivity study in Section 3.4.5.



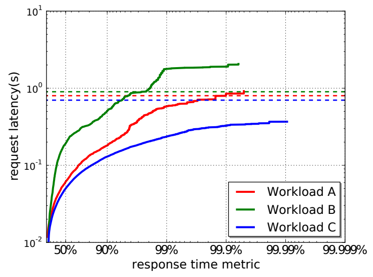
(a) PM: all workloads' SLOs satisfied



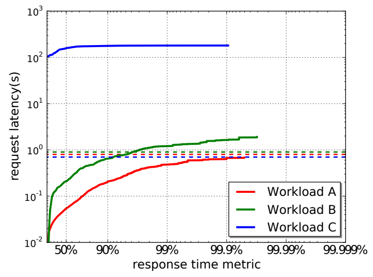
(b) Cake(reactive): blue SLO violated @ 84th percentile



(c) bySLO: green SLO violated @ 91st percentile



(d) EDF: green SLO violated @ 97th percentile



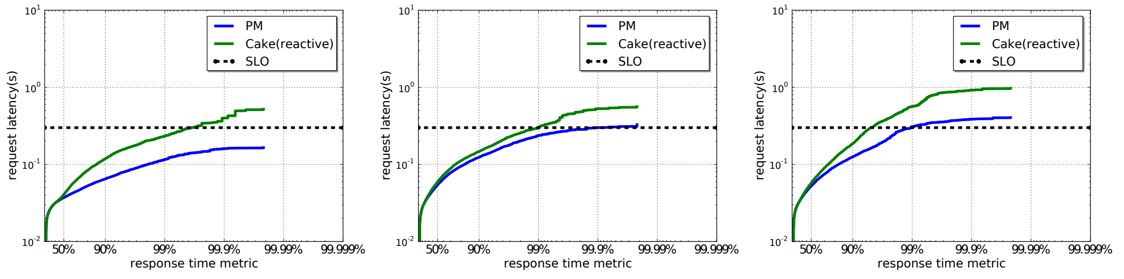
(e) PS: blue SLO always violated

Figure 3.4: Request latency at different percentiles for each policy. Same experiment as in Figure 3.3 with a more descriptive representation. It is easy to see that PriorityMeister (PM) is the only policy that doesn't violate any SLOs (dashed lines).

3.4.1 PriorityMeister tail latency performance

Figure 3.3 plots tail latency performance across multiple policies, co-located workloads, and tail latency percentiles. PriorityMeister (PM) is the only policy that meets SLOs for all workloads across all percentiles. In this experiment, we show a typical example where the storage is a bottleneck. We replay three traces derived from production workloads, combined with a throughput-oriented workload (Workload L; not shown) that represents a file copy, all sharing a single disk. All policies satisfy the throughput requirement, but not all policies meet latency SLOs (dashed lines), especially at high percentiles.

Figure 3.4 shows a more descriptive representation of the latency (y-axis) at different percentiles (x-axis). It is essentially a representation of the CDF in log scale to focus on the tail behavior, with higher percentiles on the right. The results are grouped by scheduling policy, and it is easy to see that PriorityMeister is the only policy that doesn't violate any SLOs.



(a) low burstiness, $C_A^2 = 1$, Workload F

(b) high burstiness, $C_A^2 = 20$, Workload G

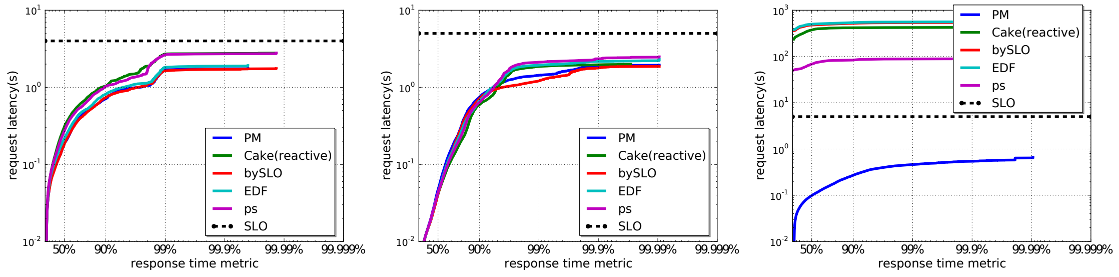
(c) very high burstiness, $C_A^2 = 40$, Workload H

Figure 3.5: Effect of burstiness on tail latency. Increased levels of burstiness affect both PriorityMeister (PM) and Cake, but PM meets the SLO at the 99th percentile for inter-arrival burstiness levels up to $C_A^2 = 40$.

So why do the other policies fail? Proportional sharing (ps) is a strawman example of not using latency QoS and is expected to fail. Cake [76] suffers from a combination of three effects. First, reactive algorithms by design only react to problems. These approaches do not work when targeting higher tail latencies where we cannot miss SLOs. Second, the burstiness found in production traces exposes the aforementioned shortcomings of reactive approaches. Third, there are more parameters to dynamically adjust when co-locating more than one latency sensitive workload. Since the workloads are bursty at potentially different times, it is not clear whether the parameters will even converge. Although the Cake paper only targets a single latency sensitive workload, we were hoping that it could be generalized to a few workloads, but we were unable to successfully do so with our workloads. EDF and bySLO are both policies that only take into account the SLO of a workload. By not considering the burstiness and load of workloads, they sometimes make bad prioritization decisions. For example, bySLO prioritizes Workload C, which has a high load that has a large impact on the other workloads. PriorityMeister accounts for the load and burstiness of workloads to determine better priority orders as seen in this experiment.

3.4.2 Coping with burstiness

In Figure 3.5, we perform a micro-benchmark on the effect of burstiness on proactive (PriorityMeister) and reactive (Cake) approaches. As burstiness increases, it is harder to meet SLOs for all policies, but our proactive approach consistently does better. To make a fairer comparison between these approaches, we only use a single latency sensitive workload and throughput-oriented workload as in the Cake paper [76]. To vary the burstiness of a



(a) Workload D, behaved

(b) Workload C, behaved

(c) Workload C when Workload D misbehaves

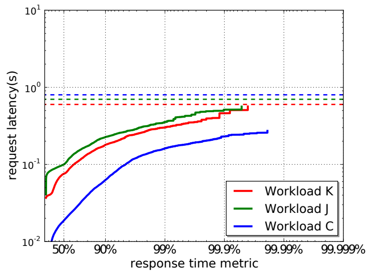
Figure 3.6: Prioritization is safe with rate limiting. PriorityMeister is the only policy that isolates the effect of the misbehaving Workload D on Workload C.

workload, we synthetically generate random access traces where we control the distribution of inter-arrival times. As a reference point, we do see that Cake meets the 99th percentile for the low burstiness trace. However, as the burstiness increases, Cake is unable to meet SLOs, even at the 96th percentile. PriorityMeister is better able to cope with the burstiness by prioritizing the latency sensitive workload, though there are cases (e.g., Figure 3.5(c)), as expected, where it is not possible to meet SLOs at the tail. This is because burstiness inherently increases the queueing and latency of a workload.

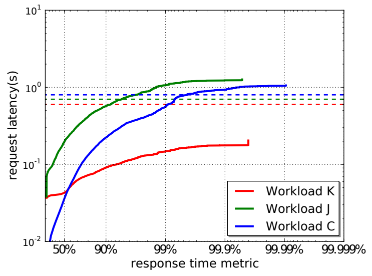
3.4.3 Misbehaving workloads

Since PriorityMeister is automatically configuring workload priorities, a natural question to ask is whether prioritization is safe. If a workload misbehaves and hogs the bandwidth, a good QoS system is able to contain the effect and avoid starving the other well-behaved workloads. PriorityMeister solves this by using rate limiting.

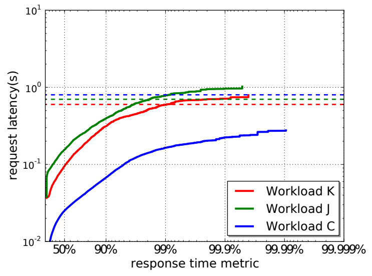
Figure 3.6 demonstrates the effect of rate limiting with a two workload scenario where Workload D, configured with a high priority, changes from being well-behaved to misbehaved (Workload E). We set the SLOs to be high enough for *all* policies to meet them under normal conditions (Figure 3.6(a), Figure 3.6(b)). However, when Workload D misbehaves and floods the system with requests, Workload C, at a lower priority, (Figure 3.6(c)) is negatively impacted. PriorityMeister is the only policy that manages to limit the effect of the misbehaving Workload D through its rate limiting. This demonstrates that prioritization is safe with rate limiting since Workload D has a higher priority in this experiment.



(a) PM: all workloads' SLOs satisfied



(b) bySLO: green SLO violated @ 95th percentile



(c) no QoS: green SLO violated @ 98th percentile

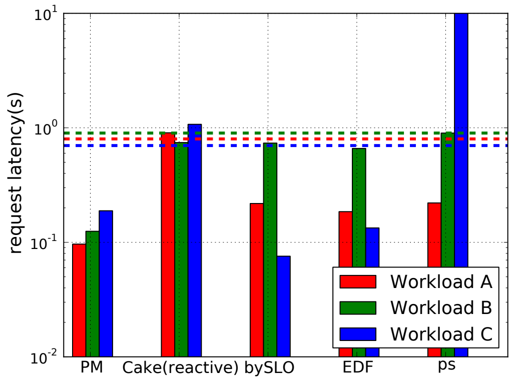
Figure 3.7: Effect of network bottleneck. When workloads induce a bottleneck on server network egress, PM is the only policy that meets all SLOs across all latency percentiles.

3.4.4 Multi-resource performance

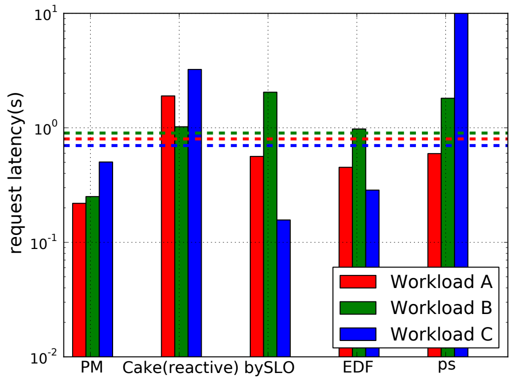
With the growing popularity of SSDs and ramdisks, the bottleneck could sometimes be the network rather than storage. PriorityMeister is designed to generalize to both network and storage and potentially other resources in the future. Since network packets can be prioritized in many network switches, PriorityMeister can operate on existing hardware.

Two common locations for a network bottleneck is at the server egress and the client ingress. In these experiments, we use a set of four workloads on servers with a ramdisk and multiple disks. To focus on the network aspect, each workload runs on a dedicated storage device. For clarity, we only show three of the workloads where there is an effect on meeting SLOs. The other workload (Workload I) has a higher SLO that is satisfied by all policies. Figure 3.7 shows an experiment with a server egress bottleneck where all the workloads are accessing storage devices co-located on a single machine. Our experimental results with a client ingress bottleneck are similar. Both scenarios motivate the need for network traffic conditioning. Without network QoS (Figure 3.7(c)), workloads start missing their SLOs at the tail. PriorityMeister (Figure 3.7(a)) solves this problem by prioritizing the three shown workloads in a way that is aware of both storage and network. Since Workload K is the only workload running on a ramdisk, PriorityMeister realizes that the storage requests will be fast and that it does not need to give workload K the highest network priority. By contrast, the bySLO policy (Figure 3.7(b)) simply gives workload K the highest priority because it has the lowest SLO, causing SLO violations at the tail latencies of other workloads.

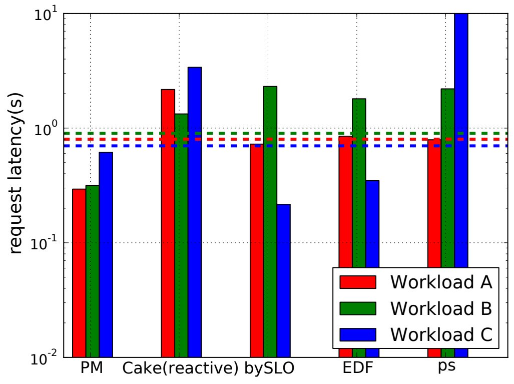
We only show three policies in these experiments since EDF and Cake do not generalize to networks. EDF would require a mechanism to timestamp packets and order packets by timestamp, which is not supported in network switches. Cake would require a mechanism to proportionally share the network, which is difficult to do in a distributed environment.



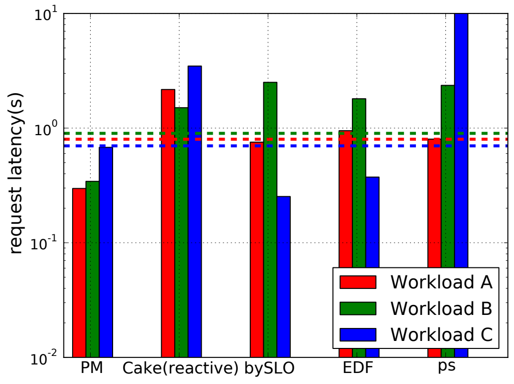
(a) 90th percentile



(b) 99th percentile



(c) 99.9th percentile



(d) 99.99th percentile

Figure 3.8: PriorityMeister is robust to storage latency mis-estimation. Same experiment as Figure 3.3, but with a less accurate storage model. Despite the mis-estimation, results are similar.

3.4.5 Sensitivity analysis

Storage model inaccuracy

Storage modeling is known to be a challenging problem, and we are interested in how well PriorityMeister performs when our storage model (described in Section 2.3.2) is inaccurate. To demonstrate the robustness to modeling inaccuracy, we replace our default storage model with a simple model that assumes a constant seek time for servicing a request. Figure 3.8 shows the same experiment as in Figure 3.3, but with the less accurate model. We find that PriorityMeister ends up selecting the same priority order and producing similar results.

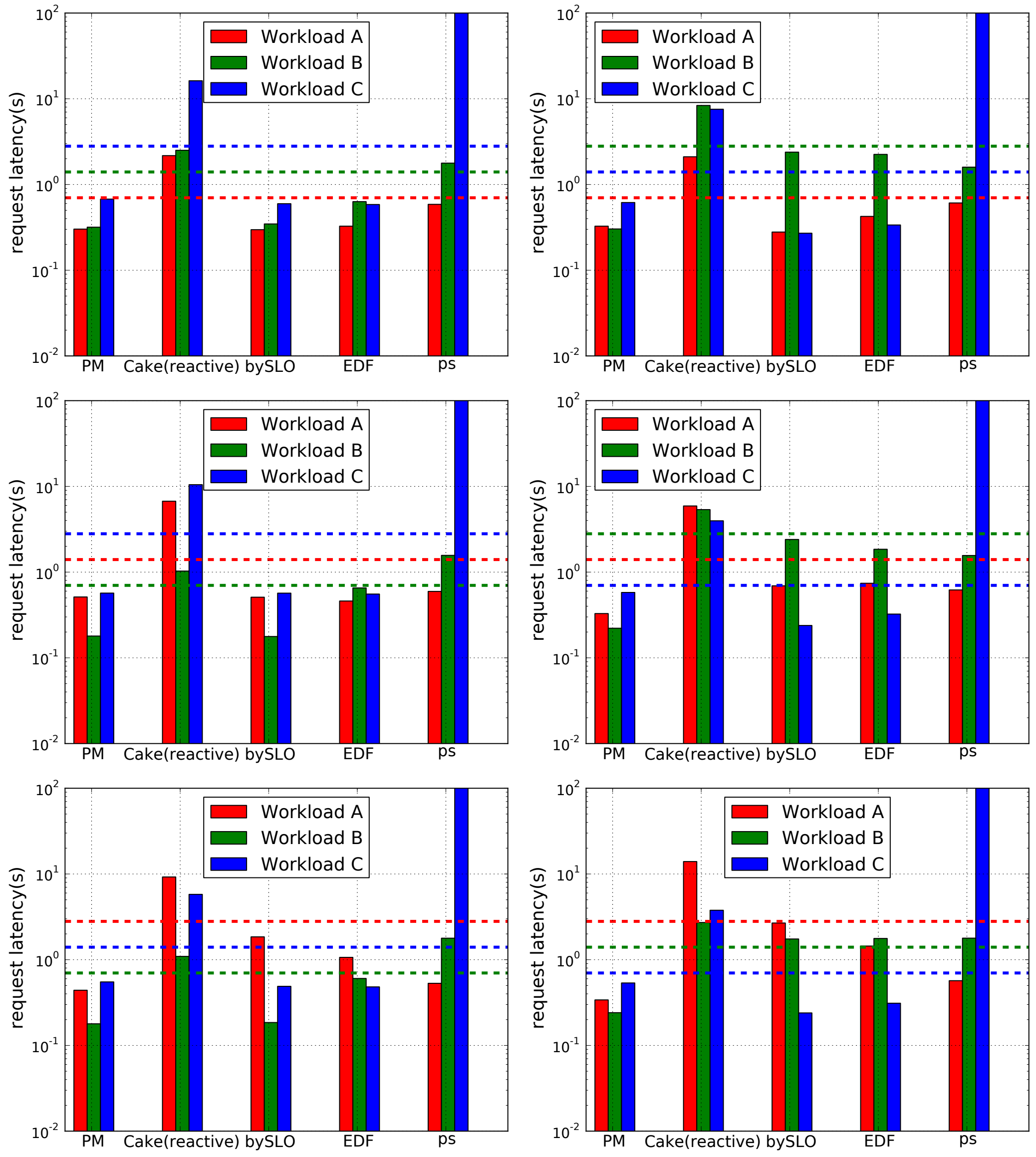


Figure 3.9: Effect of selecting different SLOs. The 6 graphs each show a different permutation of selecting 3 SLO values. 99.9th percentile latency bar plots are shown for each permutation. PM meets all SLOs for all permutations, while other policies do not. Note that bySLO does surprisingly well, meeting SLOs in 5 of the 6 experiments.

Scheduler	Latency SLO	Multi-resource
Argon [75]	No	No
SFQ(D) [45]	No	No
AQuA [79]	No	No
mClock [38]	No	No
PARDA [36]	No	Yes
PISCES [67]	No	Yes
Maestro [56]	Average latency	No
Triage [46]	Average latency	No
Façade [55]	Average latency	No
pClock [35]	Average latency	No
Avatar [85]	95th percentile	No
Cake [76]	99th percentile	Yes
PriorityMeister	$>$ 99th percentile	Yes

Table 3.2: Comparison of storage schedulers.

SLO variation

The choice of SLO is dictated by the user and will certainly have an impact on how the policies perform. We are interested to see how different SLOs affect PriorityMeister in comparison to the other policies. To do this, we rerun the same experiment as in Section 3.4.1 but with different SLOs for the workloads. Motivated by the bySLO policy, we pick three SLO numbers and try the 6 (= 3!) permutations for assigning the SLOs to workloads. Figure 3.9 shows the 99.9% latencies for these experiments. PriorityMeister meets SLOs in all 6 experiments. Surprisingly, we find that the simple bySLO policy does a reasonable job at meeting SLOs for 5 of the 6 experiments. While bySLO does not meet SLOs in as many cases as PriorityMeister, it is a decent heuristic, especially in cases where SLO values differ significantly.

3.5 Related work

PriorityMeister is different from prior work in two main ways. First, it is designed specifically for meeting tail latency SLOs in multi-tenant storage environments. Second, PriorityMeister generalizes to multiple resources including network and storage. Table 3.2 compares existing schedulers and PriorityMeister.

Tail latency

Most of the prior work on storage scheduling has focused on the easier problem of sharing storage bandwidth [36, 38, 45, 67, 75, 79]. Of the ones that focus on latency, most of them target the average latency [35, 46, 55, 56]. We are only aware of two storage schedulers, Cake [76] and Avatar [85], that investigate tail latency behavior.

Cake [76] is a reactive feedback-control scheduler that adjusts proportional shares to meet 99th percentile latency SLOs. Our goals are similar, but PriorityMeister employs a different approach to overcome some of Cake's limitations. Cake only handles one latency-sensitive workload with one throughput-oriented workload. PriorityMeister can handle multiple latency and throughput SLOs, and it automatically tunes all of its system parameters. Furthermore, PriorityMeister can deal with the burstiness found in production storage traces, and it can meet higher percentile latency SLOs (e.g., 99.9%), both of which are not possible using a reactive approach.

While Cake addresses multiple resources for HBase (CPU) and HDFS (storage), it requires a mechanism to dynamically adjust proportional shares that is not readily available for networks. Instead, PriorityMeister uses priority, which is a much simpler mechanism and has support in many network switches. We tried extending Cake to use network rate limits as a proxy for proportional shares, but it turned out to hurt more than help.

Avatar [85] is an Earliest Deadline First (EDF) scheduler with rate limiting support. While Avatar shows tail latency performance, only the 95th percentile is evaluated (in simulation). Our work focuses on higher tail latencies (e.g., 99.9%), and we perform our evaluation on actual hardware. Avatar finds that rate limiting is important for providing performance isolation, but it does not address how to set the rate limits, and its rate limiting model is not configurable for workloads of varying burstiness. PriorityMeister analyzes workload traces to automatically configure rate limits, and it can work with workloads of varying burstiness. Lastly, the focus in Avatar is solely on storage, and the solution does not generalize to networks since EDF relies on having a single entity that can timestamp and order requests.

Multi-resource

A few recent papers have started to investigate the challenges with multi-resource scheduling [32, 33, 36, 67, 71, 76]. Providing QoS across multiple resources is particularly relevant for end-to-end latency SLOs since latency is cumulative across all the resource stages (e.g., storage, CPU, network, etc). One could imagine using two different QoS systems for storage and network, but it is not obvious how to determine SLOs for each stage based on a given total end-to-end SLO. PriorityMeister is a single QoS system that understands both storage and network and can automatically configure the system to meet end-to-end latency

SLOs. Our multi-resource QoS architecture is most similar to that of IOFlow [71]. IOFlow introduces a new software-defined storage architecture for both storage and network QoS, but does not address how to configure the system to meet latency SLOs. Our work is complementary to IOFlow and can be thought of as a policy that could be built on top of IOFlow's architecture.

Other related work

The Bobtail [81] paper also investigates the problem of tail latencies in the cloud, and the authors find a root cause of bad CPU co-scheduling. Our work is complementary to theirs, and our work has the potential of incorporating CPU QoS in the future. HULL [3] addresses the problem of delays from long network switch queues by rate limiting and shifting the queueing to the end-hosts. Xu et al. [80] also address this problem, but do so using network prioritization. Both papers allow low bandwidth workloads to quickly pass through the network switch, but do not address how to deal with higher bandwidth workloads with different end-to-end latency SLOs. PriorityMeister draws upon the field of deterministic network calculus for modeling workloads and uses concepts such as arrival curves and service curves [50]. Our latency analysis is similar to a recent theory paper by Bouillard et al. [11].

3.6 Chapter summary

This chapter looks at how to meet tail latency SLOs in a shared networked storage system. We find that existing reactive approaches are unable to cope with the burstiness found in production workloads. Tail latency is significantly impacted by the bursts before a reactive approach can react to the problem.

Our solution, PriorityMeister, takes a different approach by incorporating a tail latency calculator that calculates the queueing effects from workload burstiness. PriorityMeister uses this calculator to automatically configure priorities and rate limits to meet tail latency SLOs. Experiments with production workload traces on a real system show cases where PriorityMeister can meet even extreme tail latency SLOs (e.g., 99.99%), whereas state-of-the-art approaches cannot.

PriorityMeister's tail latency calculator is based on the Deterministic Network Calculus (DNC) theory, which is a powerful tool for analyzing worst-case latency in a network of queues. Since the publication of PriorityMeister, two other network QoS systems, Silo [43] and QJump [34], have adopted DNC-based approaches. Beyond storage and networks, PriorityMeister's techniques could also be extended to analyzing latency in real-time systems, where prioritization is common and strict guarantees are desired.

One may be concerned about the worst-case nature of the DNC analysis. A worst-case analysis is certainly applicable to settings where workloads may be correlated or adversarial. But in cases where workloads are generally independent, the DNC worst-case analysis can be overly conservative. In the next chapter, we provide a solution, SNC-Meister, for a more accurate latency analysis with non-adversarial workloads. Thus, PriorityMeister and SNC-Meister collectively cover the spectrum from potentially correlated/adversarial workloads to uncorrelated/independent workloads.

Chapter 4

SNC-Meister: Admitting more workloads with tail latency SLOs

In this chapter, we focus on the question of how to perform admission control for tail latency SLOs. When a new workload arrives to the system, we want to decide at that moment whether we can admit the workload into the system while still guaranteeing all existing SLOs and the new workload's SLO. The key challenge lies in determining upper bounds on each workload's tail latency, which is particularly challenging with bursty workloads.

While our prior work, PriorityMeister, does not discuss admission control, the Deterministic Network Calculus (DNC) analysis used in PriorityMeister could be applied to admission control. However, DNC is a worst-case queueing analysis targeting the 100th percentile latency. In practice, many workloads are satisfied with controlling lower SLO percentiles such as the 99.9th and 99th percentiles. As we'll see in our results, PriorityMeister [86] and other DNC-based systems [34, 43] are too conservative in admitting workloads with lower SLO percentiles. This is because as a worst-case analysis, DNC fundamentally must account for every scenario, including adversarial worst-case scenarios where all the workloads have their worst bursts at exactly the same time. By contrast, SLO guarantees at lower percentiles do not need to cover these unrealistic worst-case scenarios.

We present SNC-Meister, a new approach for handling SLOs at various tail latency percentiles. The key difference in SNC-Meister is that it uses a new probabilistic analysis technique called Stochastic Network Calculus (SNC), which can analyze any latency percentile (e.g., 99%). Until now, SNC has only been studied in theory, and SNC-Meister is the first computer system to apply this new branch of theory. Focusing on tail latency percentiles, rather than the adversarial worst-case DNC latency, allows SNC-Meister to admit many more workloads: in experiments with production traces, SNC-Meister supports 75% more workloads than the state-of-the-art while meeting tail latency SLOs.

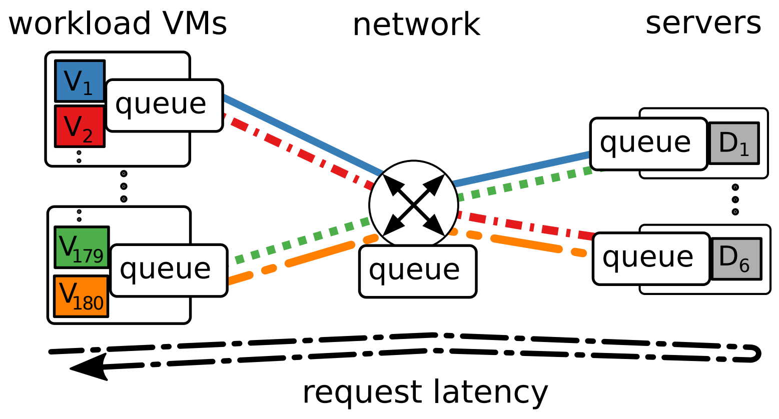


Figure 4.1: SNC-Meister meets tail latency SLOs for workloads in the network shown. Our evaluation experiments involve six servers running memcached and 180 workload VMs (on 12 machines), which replay recent production traces.

We introduce the problem and discuss the scope of this chapter in Section 4.1. We present the design and implementation of SNC-Meister in Section 4.2. We then describe our experimental setup in Section 4.3 followed by our results in Section 4.4. We discuss related work in Section 4.5 and conclude with a summary of this chapter in Section 4.6.

4.1 Introduction

Meeting tail latency Service Level Objectives (SLOs) in multi-tenant cloud environments is challenging. A tail latency SLO such as a 99th percentile of 50ms requires that 99% of requests complete within 50ms. Researchers and companies such as Amazon and Google repeatedly stress the importance of achieving tail latency SLOs at the 99th and 99.9th percentiles [5, 20, 21, 34, 42, 43, 62, 70, 73, 74, 81, 86]. As demand for interactive services increases, the need for latency SLOs will become increasingly important. Unfortunately, there is little support for specifying tail latency requirements in the cloud. Latency is hard to guarantee since it is affected by the burstiness of each workload and the congestion between them. Tail latency is particularly affected by burstiness, and recent measurements show that the 99.9th latency percentile can be high and vary tremendously [57].

The case for request latency SLOs

In this chapter, we consider cloud workloads that issue a series of requests over time for data items on another server VM within the same datacenter. For example, in Figure 4.1, the blue workload, residing on VM V_1, sends requests to server VM D_1, which hosts its data. We define SLOs over a pair of VMs (e.g., (V_1, D_1)), which is known in literature as the pipe model. We define SLOs in terms of *request latency* (a.k.a. flow completion time), which is the total time from when a workload issues a request until *all* the requested data is received. Request latency is different from *packet latency*, which is the time it takes a single packet to traverse through the network. Packet latency is the right metric when requests are small and load is light. However, as the amount of data used increases, request latency becomes the most relevant granularity (as argued in [84]).

Queueing is inevitable for request latency

High request latency is almost always due to excessive queueing delay [34, 43]. Queueing is inevitable. In production environments, traffic is typically bursty. When these bursts happen simultaneously, the result is high queueing delays.

Queueing can occur both within the network (*in-network queueing*) and at the end-hosts (*end-host queueing*). Some works (e.g., Fastpass [62], HULL [4]) claim to eliminate or significantly reduce queueing. What they actually mean is that they eliminate *in-network queueing* by shifting the queueing to the end-hosts with rate limiting. This produces great benefits for packet latency, which does not include this end-host queueing time. However, these techniques do not solve the problem for request latency, which by definition captures the entire queueing time, both in-network queueing and end-host queueing. Both forms of queueing delay comprise the biggest portion of request latency, particularly when looking at the tail percentiles [34]. Our system focuses solely on the effects of queueing (i.e., congestion between workloads) and leaves the mitigation of other sources of tail latency (e.g., VM scheduling, TCP artifacts) to other works (e.g., [80, 81]).

Dual goals: meeting tail latency SLOs and achieving high multi-tenancy

The goal of our system is two-fold: 1) we want to meet tail request latency SLOs; and 2) we want to admit as many workloads as possible. Clearly, there is a tradeoff between achieving both goals. Admitting few workloads will likely meet SLOs due to limited queueing. Admitting many workloads, in contrast, creates the possibility of SLO violations due to high contention between workloads. *Admission control* is the component that limits the multi-tenancy so as to guarantee that the system *only* admit workloads whose SLOs can be met.

A key challenge in admission control is predicting upper bounds on the request latency for each workload. Predicting latency bounds is only possible with assumptions on workload behavior. We address the typical behavior of workloads (i.e., not flash crowds, faulty hardware, etc). We assume that typical workload behavior can be characterized (or at least upper bounded) by a stationary trace of past behavior. A trace contains a list of requests parameterized by arrival time and request size. Figure 4.2(a) shows a graphical representation of three example traces. Our system extracts information about a workload's load and burstiness from its trace.

Note that bursts are short lived (on the order of seconds), and that these bursts are not caused by diurnal or hourly trends. In this work, we specifically focus on this short-term burstiness, which is separate from time-varying load[1]. These short-term bursts occur during every hour of our traces, and they are known to have a large impact on performance [41].

The state of the art in admission control: worst-case bounds on the request latencies

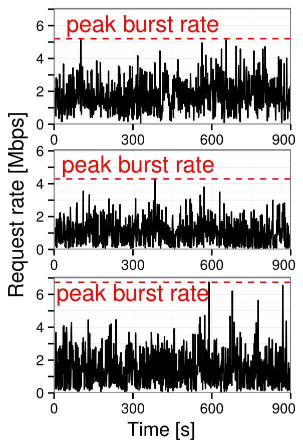
The state-of-the-art in admission control are Silo (SIGCOMM 2015 [43]), QJump (NSDI 2015 [34]), and PriorityMeister (SoCC 2014 [86]). These systems perform admission control by using Deterministic Network Calculus (DNC) to calculate upper bounds on the request latency. Typically, DNC-based systems assume a workload's request process is characterized based on a maximum arrival rate and burst size. They then use DNC to calculate each workload's worst-case latency based on the workloads' maximum rate/burst constraints. If the worst-case latency for a workload is higher than its SLO, the workload is not admitted.

The above systems all use DNC, but in somewhat different ways. Silo uses DNC to calculate the amount of queueing within the network and performs admission control to ensure that network switch buffers do not overflow. QJump offers several classes with different latency-throughput trade-offs, for which latency guarantees are calculated with DNC. PriorityMeister considers different prioritizations of workloads. For each priority ordering, PriorityMeister uses DNC to calculate the worst-case latency of each workload. PriorityMeister aims to choose a priority ordering that maximizes the number of workloads that can meet their SLOs if admitted.

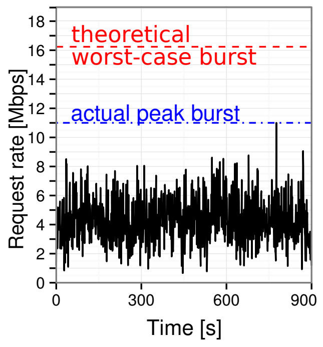
The limitations of DNC

The DNC theory predicts worst-case latencies for the adversarial case where the worst possible bursts of all workloads happen simultaneously. While some workloads may be adversarially correlated, it is very conservative to assume all workloads are correlated with each other. The difference between assuming independence and dependence is substantial; as an example, Figure 4.2(b) shows the aggregate behavior of the three traces in

[1]Time-varying load can be accommodated by using a trace from a period of high load or by updating the workload's trace over time. Our work is still relevant to the short-term burstiness that occurs during periods of high load.



(a) Individual burstiness



(b) Aggregate burstiness

Figure 4.2: Three example production traces and their aggregate trace. A worst-case analysis assumes that all three individual traces have their worst peaks at the same time, which is overly conservative as shown in the aggregate trace.

Figure 4.2(a). The peak burst in each trace is marked with a horizontal line. As DNC is an adversarial worst-case analysis technique, its equations account for the scenario where each of the peak bursts happen at the same time. But as shown in the aggregate trace, the actual peak is much lower than the adversarial sum of peaks. As a result, DNC's worst-case assumption limits the number of workloads that can be admitted into the system for any given SLOs.

The case for Stochastic Network Calculus (SNC)

Typical users do not seek strict worst-case guarantees. Instead, users target tail latency percentiles lower than the 100%, e.g., the 99.9th latency percentile [21]. DNC only supports calculating the 100th percentile latency (i.e., adversarial worst-case). So given 99.9th percentile SLOs, DNC-based systems simply pretend they are 100th percentile SLOs, resulting in admission decisions which are conservative.

We therefore instead turn to an emerging branch of probabilistic theory called Stochastic Network Calculus (SNC). SNC provides request latency bounds for any user-specified latency percentile, e.g., the 99th, 99.9th, or 99.99th latency percentile. By not making adversarial worst-case assumptions, it is possible to admit many more workloads, even for high percentiles (several 9s).

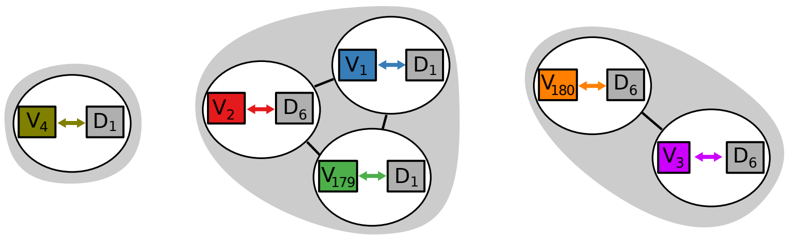


Figure 4.3: User-specified workload dependency graph with three groups. Workloads in a dependent group are assumed to be adversarially correlated with each other.

Support for dependencies in SNC

The SNC theory also supports having certain workloads being dependent on each other, as indicated by a user-specified dependency graph (Figure 4.3). A user running several related workloads can specify that a group of workloads are dependent on each other. Dependent workloads in a group are allowed to be adversarially correlated with each other, but are assumed to behave independently in relation to workloads in other groups. Thus, it is possible to capture the benefits of independence without assuming all workloads are fully independent of every other workload.

Our SNC-based system: SNC-Meister

Our new system, SNC-Meister, uses SNC to upper bound request latency percentiles in a shared system with multiple workloads. SNC-Meister makes admission decisions for the specific latency and percentile requested by each workload. In this chapter, we focus on networks, but our techniques also apply to storage as demonstrated in Section 4.4.6. We implement and run SNC-Meister on a physical cluster, and our experiments with production traces show that SNC-Meister can support many more workloads than the state-of-the-art systems by considering 99.9th percentile SLOs (see Figure 4.4).

In this chapter, we make the following main contributions:

- **Bringing SNC to practice:** SNC is a new theory that has been developed in a purely theoretic context and has never been implemented in a computer system. Our primary contribution is identifying and overcoming multiple practical challenges in bringing SNC to practice. For example, it is an open problem how to effectively apply SNC in non-trivial network topologies and how to incorporate workload dependencies. We prove the correctness of SNC-Meister's analysis and show that SNC-Meister improves the tightness of SNC latency bounds by 2-4×.
- **Extensive evaluation:** We implement SNC-Meister and evaluate it on an 18-machine

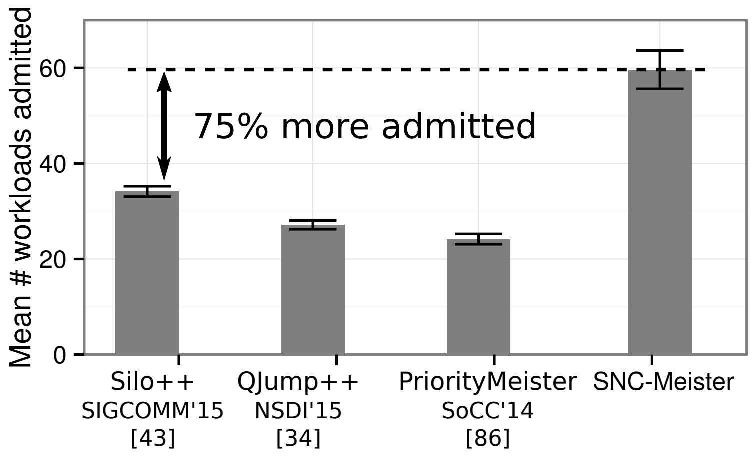


Figure 4.4: Admission numbers for state-of-the-art admission control systems and SNC-Meister in 100 randomized experiments. In each experiment, 180 workloads, each submitting hundreds of thousands of requests, arrive in random order and seek a 99.9% SLO randomly drawn from {10ms, 20ms, 50ms, 100ms}. While all systems meet all SLOs, SNC-Meister is able to support on average 75% more workloads with tail latency SLOs than the next-best system.

cluster running the widely-used memcached key-value store (setup shown in Figure 4.1, details in Section 4.3). We compare against three state-of-the-art admission control systems, two of which we enhance to boost their performance[2]. Across 100 experiments each with 180 workloads represented by recent production traces, SNC-Meister is able to support on average 75% more workloads than the enhanced state-of-the-art systems (Figure 4.4) while meeting SLOs of all admitted workloads. This improvement means that SNC-Meister allows workloads to transfer 88% more bytes in the median (Section 4.4.1). SNC-Meister is also within 7% of an empirical offline maximum, which we determined through trial-and-error experiments (Section 4.4.2).

- **Open-source release of SNC-Meister:** Code for SNC-Meister is available at `https://github.com/timmyzhu/SNC-Meister`. We design SNC-Meister to operate in existing infrastructures alongside best effort workloads without requiring kernel, OS, or application changes. To simplify user adoption, SNC-Meister only requires high-level user input (e.g., SLO, trace) and automatically generates SNC models

[2]Silo++ admits 10% more workloads than a hand-tuned Silo baseline, and QJump++ admits 5× more workloads than a hand-tuned QJump baseline.

and corresponding configuration parameters. Our representation of SNC in code is simple and efficient, which results in the ideal linear scaling of computation time for admission decisions in terms of the number of workloads.

4.2 SNC-Meister

In determining admission, SNC-Meister works with per-workload tail latency SLOs and traces representing the burstiness and load added by each workload. Traces consist of a sequence of request arrival times and sizes, and they can be extracted from historical logs or captured during operation. Using traces avoids the burden of having users specify many complex parameters to describe their traffic.

After receiving a workload's SLO and trace, SNC-Meister determines admission through the following three steps. First, SNC-Meister analyzes the workload's trace to derive a statistical characterization understood by the SNC theory (see Section 4.2.4). Second, SNC-Meister assigns a priority to the workload based on its SLO where the highest priorities are assigned to workloads with the tightest SLOs (i.e., lowest latency value). We opt for this simple prioritization scheme since our experiments with PriorityMeister's more complex prioritization scheme show similar results. Third, SNC-Meister calculates the latency for each workload based on SNC (see Section 4.2.2) and checks if each workload's predicted latency is less than its SLO. If the previously admitted workloads and the new workload all meet their SLOs, then the new workload is admitted at its priority level. Otherwise, the workload is rejected and can only run at the lowest priority level as best-effort traffic. SNC-Meister enforces priorities both in switches and at end-hosts as described in Chapter 2.

We next provide background on SNC (Section 4.2.1) followed by four challenges we overcome in implementing SNC-Meister:

1. SNC is a new theory, and it is currently an open problem how to effectively apply SNC to network topologies (e.g., Figure 4.1). The SNC literature is primarily concerned with theorems and proofs, but little is known about applying them in practice. Section 4.2.2 describes SNC-Meister's novel network analysis technique and the corresponding improvement in accuracy.
2. The SNC literature does not consider the analysis of dependencies between workloads. Section 4.2.3 discusses how SNC-Meister handles dependencies and its effect on latency.
3. Real traffic exhibits bursty behavior, particularly at second/sub-second granularities, and it is important to capture this behavior to properly characterize tail latency. Section 4.2.4 describes how SNC-Meister models burstiness and how it configures model parameters based on trace data.

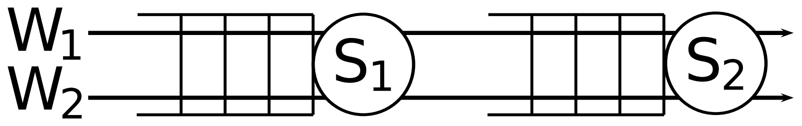


Figure 4.5: Example network with two workloads W_1 and W_2 flowing through two queues S_1 and S_2.

4. It is non-trivial how to work with full representations of probabilistic distributions in software as required by SNC. Section 4.2.5 describes how SNC-Meister is implemented in code.

4.2.1 Stochastic Network Calculus background

At the heart of SNC-Meister is the Stochastic Network Calculus (SNC) calculator. SNC is a mathematical toolkit for calculating upper bounds on latency at any desired percentile (e.g., 99th percentile). This is in contrast to DNC, which computes an upper bound on the worst-case latency (i.e., 100th percentile). We next describe the core concepts of SNC by way of example (Figure 4.5) followed by the necessary mathematical details needed to implement SNC.

SNC core concepts

SNC is based on a set of operators that manipulate probabilistic distributions. We refer to these distributions as *arrival processes* (A_1 and A_2 for workloads W_1 and W_2 in Figure 4.5) and *service processes* (S_1 and S_2 in Figure 4.5). One of the main results from SNC is a *latency operator* for taking an arrival process (e.g., A_1), a service process (e.g., S_1), and a percentile (e.g., 0.99), and calculating an upper bound on the tail latency. We write this as $Latency(A_1, S_1, 0.99)$. The latency operator works for any arrival and service processes.

As an example, consider calculating the 99th percentile latency for W_1 in Figure 4.5. Since W_1 and W_2 share the first queue, W_1 does not experience service process S_1 since there is congestion introduced by W_2. Rather, W_1 experiences the leftover (a.k.a. residual) service process after accounting for W_2. In SNC, this is handled by the *leftover operator*, $\ominus$, which is used in our example to calculate a new service process $S_1' = S_1 \ominus A_2$. W_1's 99th percentile latency at the first queue is then calculated by using the latency operator with S_1' (i.e., $Latency(A_1, S_1', 0.99)$).

Calculating W_1's latency at the second queue in Figure 4.5 requires arrival processes at the second queue, which are precisely the output (a.k.a. departure) processes from the first queue. In SNC, this is handled by the *output operator*, $\oslash$, which is used in our example to

Purpose	$\rho(\cdot)$	$\sigma(\cdot)$
Arrival process A for MMPP with transition matrix Q and diagonal matrix $E(\theta)$ of each state's MGF	$\rho_A(\theta) =$ $sp(E(\theta)\,Q)$	$\sigma_A(\theta) =$ 0
Service process S for network link with bandwidth R	$\rho_S(\theta) =$ $-R$	$\sigma_S(\theta) =$ 0
Leftover operator $\ominus$ for service process S and arrival process A	$\rho_{S\ominus A}(\theta) =$ $\rho_A(\theta) + \rho_S(\theta)$	$\sigma_{S\ominus A}(\theta) =$ $\sigma_A(\theta) + \sigma_S(\theta)$
Output operator $\oslash$ for service process S and arrival process A	$\rho_{A\oslash S}(\theta) =$ $\rho_A(\theta)$	$\sigma_{A\oslash S}(\theta) =$ $\sigma_A(\theta) + \sigma_S(\theta) -$ $\frac{1}{\theta}\log\left(1 - e^{\theta(\rho_A(\theta)+\rho_S(\theta))}\right)$
Aggregate operator $\oplus$ for arrival process A_1 and arrival process A_2	$\rho_{A_1\oplus A_2}(\theta) =$ $\rho_{A_1}(\theta) + \rho_{A_2}(\theta)$	$\sigma_{A_1\oplus A_2}(\theta) =$ $\sigma_{A_1}(\theta) + \sigma_{A_2}(\theta)$
Convolution operator $\otimes$ for service process S_1 and service process S_2	$\rho_{S_1\otimes S_2}(\theta) =$ $\max\{\rho_{S_1}(\theta), \rho_{S_2}(\theta)\}$	$\sigma_{S_1\otimes S_2}(\theta) =$ $\sigma_{S_1}(\theta) + \sigma_{S_2}(\theta) -$ $\frac{1}{\theta}\log\left(1 - e^{-\theta\left\|\rho_{S_1}(\theta)-\rho_{S_2}(\theta)\right\|}\right)$
Tail latency L for percentile p, arrival process A, and service process S	$L =$ $\min\limits_{\theta>0} \frac{\log\left((1-p)\cdot\left(1-e^{\theta(\rho_A(\theta)+\rho_S(\theta))}\right)\right)}{\theta\rho_S(\theta)} - \frac{\sigma_A(\theta)+\sigma_S(\theta)}{\rho_S(\theta)}$	

Table 4.1: The SNC operators and equations used by SNC-Meister for independent workloads.

calculate W_1's output process, A'_1, as $A'_1 = A_1 \oslash S'_1$ where S'_1 is as defined above. W_2's output process, A'_2, is calculated similarly. W_1's latency at the second queue is then calculated as $Latency(A'_1, S_2 \ominus A'_2, 0.99)$.

One might try to calculate W_1's total latency by adding up the latencies from each queue (i.e., $Latency(A_1, S'_1, 0.99) + Latency(A'_1, S_2 \ominus A'_2, 0.99)$). However, this is not a 99th percentile latency anymore. To get a 99th percentile overall latency, higher percentiles are needed for each queue (e.g., 99.5th percentile)[3]. There are in fact many options for percentiles at each queue (e.g., 99.5 & 99.5; 99.3 & 99.7; 99.1 & 99.9) for calculating an overall 99th percentile latency. Choosing the option that provides the best latency bound is time consuming, so SNC provides a *convolution operator*, $\otimes$, which avoids this problem by treating a series of queues as a single queue with a merged service process. In

[3]This is formally known as the union bound.

our example, the convolution operator is applied to W_1's leftover service process at each queue as $S'_1 \otimes (S_2 \ominus A'_2)$. This new service process is then used to calculate W_1's latency as $Latency(A_1, S'_1 \otimes (S_2 \ominus A'_2), 0.99)$.

Lastly, SNC has an *aggregation operator*, $\oplus$, which calculates the multiplexed arrival process of two workloads. For example, the aggregate operator can be used to analyze the multiplexed behavior of W_1 and W_2 as $A_1 \oplus A_2$.

The SNC literature provides this set of operators along with proofs of correctness. However, little is known on how to best combine these operators together to analyze networks, and this is a challenging open problem that we address in SNC-Meister.

Mathematics behind SNC

We proceed to expand upon the high level description of the SNC concepts and describe the mathematics behind SNC. To begin, we define the arrival process of a workload W_1 as $A_1(m,n)$, which represents the number of bytes sent by W_1 between time m and n. As arrival processes are probabilistic in nature, SNC is based on moment-generating functions (MGFs), which are an equivalent representation of distributions. Directly working with MGFs is unfortunately quite challenging mathematically, so SNC operates on an upper bound on the MGF, parameterized by two sub-components $\rho(\theta)$ and $\sigma(\theta)$. For example, the MGF of $A_1(m,n)$, written $MGF_{A_1(m,n)}(\theta)$, is upper bounded by:

$$MGF_{A_1(m,n)}(\theta) \leq e^{\theta(\rho_{A_1}(\theta)(n-m)+\sigma_{A_1}(\theta))} \quad \forall \theta > 0$$

MGFs are parameterized by a variable θ to represent all moments of a distribution (e.g., $A_1(m,n)$). All arrival processes are specified in terms of the two sub-components $\rho(\theta)$ and $\sigma(\theta)$, and all SNC operators provide equations for these sub-components (see Table 4.1 for an overview and Appendix A.2 for full details).

To calculate the $\rho_{A_1}(\theta)$ and $\sigma_{A_1}(\theta)$ for W_1, we need to assume a stochastic process for W_1, such as a Markov Modulated Poisson Process (MMPP) (see Section 4.2.4). A MMPP is useful for representing bursty arrival rates. For example, a 2-MMPP switches between high-rate phases and low-rate phases using a Markov process. The MMPP's transition matrix is given by Q, which for a 2-MMPP has four entries:

$$Q = \begin{pmatrix} p_{hh} & p_{hl} \\ p_{lh} & p_{ll} \end{pmatrix}$$

where, e.g., p_{hl} indicates the probability of switching from a high-rate phase (h) to a low-rate phase (l). The distribution of the arrival rate and request size for each phase is captured in the matrix E, which is a diagonal matrix of the MGF for each phase:

$$E(\theta) = \begin{pmatrix} MGF_h(\theta) & 0 \\ 0 & MGF_l(\theta) \end{pmatrix}$$

Finally, the $\rho_{A_1}(\theta)$ and $\sigma_{A_1}(\theta)$ for W_1 is calculated as:

$$\rho_{A_1}(\theta) = sp(E(\theta) \cdot Q) \text{ and } \sigma_{A_1}(\theta) = 0$$

where $sp(\cdot)$ is the spectral radius of a matrix.

Service processes are defined similarly to arrival processes with the same two subcomponents $\rho(\theta)$ and $\sigma(\theta)$. Rather than working with lower bounds on the amount of service provided, SNC works with an upper bound:

$$MGF_{S_1(m,n)}(-\theta) \leq e^{\theta(\rho_{S_1}(\theta)(n-m)+\sigma_{S_1}(\theta))} \quad \forall \theta > 0$$

where the MGF has an extra negative sign on the θ parameter, which transforms the lower bound into an upper bound. For lossless networks, the $\rho_{S_1}(\theta)$ and $\sigma_{S_1}(\theta)$ have a simple form:

$$\rho_{S_1}(\theta) = -R \text{ and } \sigma_{S_1}(\theta) = 0$$

where R is the bandwidth of the network link.

Lastly, tail latency is calculated by chaining together the equations in Table 4.1 based how the SNC operators are combined (Section 4.2.2) and using the latency equation (last line in Table 4.1). Section 4.2.5 describes how we represent arrival and service processes in code and how we evaluate the latency equation with the θ parameter.

4.2.2 Analyzing networks with SNC-Meister

Analyzing networks with SNC requires an algorithm for combining the SNC operators described in Section 4.2.1. Even with the simple example in Figure 4.5, there are multiple ways to analyze the latency for W_1. For example, Section 4.2.1 describes how the latency can be analyzed one queue at a time (i.e., $Latency(A_1, S'_1, 0.995) + Latency(A'_1, S_2 \ominus A'_2, 0.995)$) as well as through a convolution operator (i.e., $Latency(A_1, S'_1 \otimes (S_2 \ominus A'_2), 0.99)$). Yet there is even another approach by first applying the convolution operator on S_1 and S_2 before accounting for the congestion from W_2 (i.e., $Latency(A_1, (S_1 \otimes S_2) \ominus A_2, 0.99)$). While each approach is correct as an upper bound on tail latency, they are not equally tight. One of our key findings is that some approaches can introduce "artificial dependencies" where arrival and service processes are treated as dependent processes even though they should be independent. For example, in $Latency(A'_1, S_2 \ominus A'_2, 0.995)$, A'_1 $(= A_1 \oslash (S_1 \ominus A_2))$ and A'_2 $(= A_2 \oslash (S_1 \ominus A_1))$ are artificially dependent because they are both derived from common sources A_1, A_2, and S_1. Likewise, the convolution $S'_1 \otimes (S_2 \ominus A'_2)$ has an artificial dependency because S'_1 $(= S_1 \ominus A_2)$ and A'_2 are both derived from S_1 and A_2. In reality, there shouldn't be any dependencies between A_1, A_2, S_1, and S_2, but the ordering of SNC

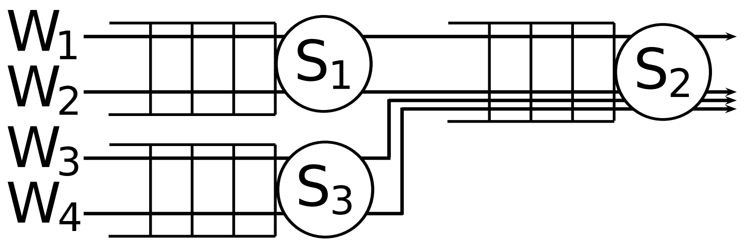


Figure 4.6: Extending Figure 4.5's example with workloads W_3 and W_4 flowing through queues S_3 and S_2.

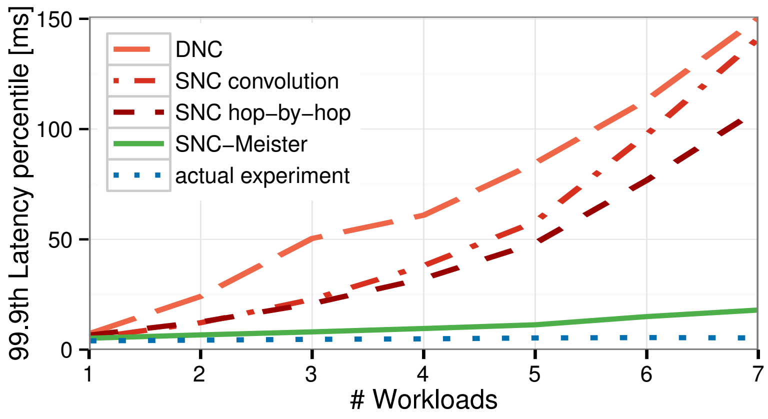


Figure 4.7: The tail latency calculated using DNC and multiple SNC methods, SNC convolution [25], SNC hop-by-hop [10], and SNC-Meister. In this micro-experiment, we vary the number of workloads connecting from a single client to a single server through two queues.

operators can introduce these artificial dependencies. A more comprehensive example for artificial dependencies can be found in Appendix A.3.

In our SNC algorithm, we identify two key ideas that allow us to eliminate artificial dependencies.

Key idea 1. When analyzing W_1, SNC-Meister performs the convolution operator before the leftover operator for any workloads sharing the same path as W_1. For example, $Latency(A_1, (S_1 \otimes S_2) \ominus A_2, 0.99)$.

Using this idea in the Figure 4.5 example avoids the artificial dependencies at the second queue. However, there are other sources of artificial dependencies. Figure 4.6 shows a slightly more complex scenario with additional traffic from W_3 and W_4. Calculating

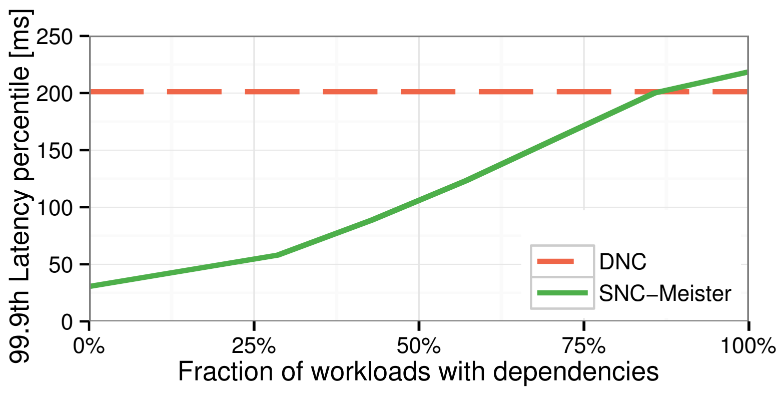


Figure 4.8: The tail latency calculated using DNC and SNC-Meister as we vary the fraction of workloads that are dependent on each other. In this micro-experiment, seven identical workloads connect from a single client to a single server, and a fraction of them (x-axis) are marked as dependent on each other.

W_1's latency now requires accounting for the effect of W_3 and W_4 at the second queue S_2. The straightforward approach is to apply the output operator on A_3 and A_4 to get arrival processes A'_3 $(= A_3 \oslash (S_3 \ominus A_4))$ and A'_4 $(= A_4 \oslash (S_3 \ominus A_3))$ at the second queue. However, this approach introduces an artificial dependency between A'_3 and A'_4 because they are derived from S_3, A_3, and A_4.

Key idea 2. When handling competing traffic from the same source, SNC-Meister applies the aggregate operator before the output operator. For example, $(A_3 \oplus A_4) \oslash S_3$.

Using this idea, the aggregate flow to the second queue now does not have any artificial dependencies. Combining the two ideas for our Figure 4.6 example, the latency of W_1 is calculated as $Latency(A_1, (S_1 \otimes (S_2 \ominus ((A_3 \oplus A_4) \oslash S_3))) \ominus A_2, 0.99)$. Through these two ideas, SNC-Meister is able to produce much tighter bounds (see Figure 4.7) than the straightforward approaches (analyzing one queue at a time: SNC hop-by-hop [10]; applying convolution to a workload's leftover service process at each queue: SNC convolution [25]). A formal description of our SNC algorithm can be found in Appendix A.4 and the proof of correctness is given in Theorem 7 in Appendix A.5.

4.2.3 Dependencies between workloads

Since not all workloads are necessarily independent, SNC-Meister also supports users specifying groups of dependent workloads. Dependent workloads are analyzed assuming they can have adversarially correlated bursts. This can be useful, for example, when multiple workloads are part of the same load balancing group.

SNC-Meister incorporates user-specified dependencies by tracking dependency information with arrival and service processes. When aggregating multiple arrival processes (as with key idea 2), SNC-Meister also uses the dependency information to minimize the number of SNC operators that assume dependence (proved in Theorem 6 in Appendix A.5).

Figure 4.8 shows the effect of workload dependency on latency. In this experiment, we take a fraction of the workloads and mark them as dependent on each other. As this fraction varies from 0% (i.e., all independent) to 100% (i.e., all dependent), we see the latency calculated by SNC-Meister increases. This is expected since dependent workloads can have higher latencies due to simultaneous bursts. Nevertheless, SNC-Meister's latency is almost always[4] under DNC since DNC assumes adversarial correlation for all workloads.

4.2.4 Modeling workload burstiness

Properly characterizing tail latency entails representing the burstiness and load that each workload contributes. In SNC-Meister, we use a Markov Modulated Poisson Process (MMPP) as an expressive and analytically tractable model for burstiness. A MMPP can be viewed as a set of phases with different arrival rates and a set of transition probabilities between the phases. A phase with high arrival rate can represent a bursty period, while a phase with low arrival rate can represent a non-bursty period. The MMPP is flexible in that the number of phases can be increased to reflect additional levels of burstiness.

The MMPP parameters for each workload are determined from its trace. Workload traces contain the arrival times of requests and their request sizes (e.g., number of bytes being requested). SNC-Meister first determines the number of MMPP phases needed to represent the range of burstiness in the trace. We use an idea similar to [40] where each phase is associated with an arrival rate and covers a range of arrival rates plus or minus two standard deviations. SNC-Meister then maps time periods in the trace to MMPP phases and empirically calculates transition probabilities between the MMPP phases.

While SNC-Meister adapts to the range of burstiness on a per-workload basis using multiple MMPP phases, the specific number of phases is not critical. In our experimentation, we find a big difference going from a single phase (i.e., a standard Poisson Process) to two phases, but less of a difference with more than two phases. If computation speed is a limiting factor, it is possible to tune SNC-Meister to compute latency faster using fewer phases.

[4]SNC-Meister can generate higher latencies than DNC when nearly all workloads are dependent because the SNC equations are not tight upper bounds, whereas our DNC analysis is tight.

4.2.5 How SNC-Meister represents SNC in code

In this section, we describe how SNC-Meister represents arrival and service processes as objects in code. We first show how to combine the SNC operators by walking through the example in Figure 4.5 and then delve into details on how SNC operators are represented internally.

To analyze the Figure 4.5 example, we start with two arrival processes (A_1 and A_2) and two service processes (S_1 and S_2):

```
ArrivalProcess* A1 = new MMPP(traceW1);
ArrivalProcess* A2 = new MMPP(traceW2);
ServiceProcess* S1 = new NetworkLink(bandwidth);
ServiceProcess* S2 = new NetworkLink(bandwidth);
```

We proceed to calculate the latency of W_1, mathematically written $Latency(A_1, (S_1 \otimes S_2) \ominus A_2, 0.99)$. First, the queues are combined to create a service process for the convolution of S_1 and S_2 (i.e., $S_1 \otimes S_2$), which is yet another service process (named `S1x2`):

```
ServiceProcess* S1x2 = new Convolution(S1, S2);
```

Second, W_1's service process is calculated by using the leftover operator on `S1x2` and W_2's arrival process (i.e., $(S_1 \otimes S_2) \ominus A_2$):

```
ServiceProcess* S1x2_A2 = new Leftover(S1x2, A2);
```

Finally, the 99th percentile latency of W_1 is calculated by:

```
double L_A1 = calcLatency(A1, S1x2_A2, 0.99);
```

SNC-Meister is designed to allow the SNC operators to compose any algebraic expression (e.g., $(S_1 \otimes S_2) \ominus A_2$ is `new Leftover(new Convolution(S1, S2), A2)`). This is accomplished by having all of the operators as subclasses of the ArrivalProcess and ServiceProcess base classes, which have a standardized representation using the $\rho(\theta)$ and $\sigma(\theta)$ form (see Section 4.2.1). To symbolically represent these $\rho(\theta)$ and $\sigma(\theta)$ functions in code, the base classes define pure virtual functions for `rho` and `sigma` that every operator overrides with the equations in Table 4.1.

Lastly, calculating latency requires optimizing the θ parameter in the Table 4.1 equations. In particular, the latency equation produces valid upper bounds on latency for every value of $\theta > 0$. Thus, to improve the accuracy of the latency bound, SNC-Meister searches for a θ that produces the minimum latency by sweeping over a range of values at a coarse granularity (e.g., $\theta = 1, 2, 3, ..., 10$) and then progressively narrowing down to finer granularities (e.g., $\theta = 2.1, 2.2, ..., 2.9$).

4.3 Experimental setup

To demonstrate the effectiveness of SNC-Meister in a realistic environment, we evaluate our implementation of SNC-Meister and three state-of-the-art systems in a physical testbed running memcached as an example application. This section describes the state-of-the-art systems (Section 4.3.1), traces (Section 4.3.2), experimental procedure (Section 4.3.3), and physical testbed (Section 4.3.4) used in our experiments.

4.3.1 Comparison approaches

We compare against three state-of-the-art systems: Silo [43], QJump [34], and PriorityMeister [86]. We enhance Silo and QJump to account for end-host queueing delay and to automatically configure workload parameters (e.g., rate limits).

Silo [43]: Silo offers workloads a worst-case packet latency guarantee under user-specified rate limits. Admission control is performed by verifying that no switch queue in the network overflows using equations from DNC. The maximum packet latency is calculated by adding up all maximum queue sizes along a packet's path.

A limitation with Silo is that choosing a rate limit (i.e., bandwidth and maximum burst size) is left to the user. In the Silo experiments, the burst size is fixed to 1.5KB, and bandwidth is chosen by trial and error. Selecting too high a bandwidth causes few workloads to be admitted. On the other hand, selecting a small bandwidth (e.g., the average bandwidth of a workload) entails a high end-host queueing delay due to being slowed down by the rate limiting. Compensating for the effect of end-host queueing is left to the user.

Silo++: We extend Silo with an algorithm to automatically choose the minimal bandwidth so that each workload's request latency SLO can be guaranteed. This is achieved by profiling each workload's traffic requirements and selecting rate limits using the effective bandwidth approach from DNC theory [51]. We also add support for calculating the end-host queueing delay using DNC, which is used in conjunction with Silo's packet latency guarantee to check whether each workload can meet its SLO.

QJump [34]: QJump offers multiple classes of service with different latency-throughput trade-offs. The first class receives the highest priority along with a 100th percentile latency guarantee using DNC-based equations [59, 60], but is aggressively rate limited. For the other classes, workloads are allowed to send at higher rates, but at lower priorities and without any latency guarantee. There are two limitations in employing the original QJump proposal: 1) users don't know which class to pick because the respective latency guarantee is unknown in advance; and 2) users don't know the end-host queueing delay caused by the rate limiting of each class.

QJump++: We extend QJump with an algorithm to automatically assign workloads to

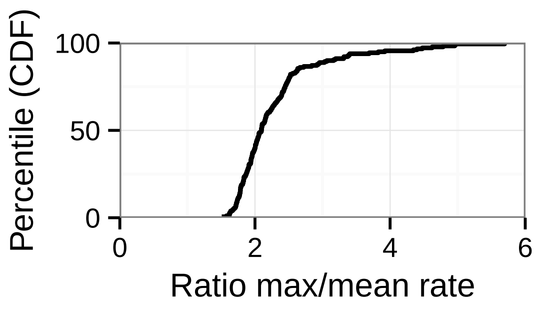


Figure 4.9: The ratio between maximum and mean request rate is high for many of our traces.

a (near) optimal class. The algorithm iteratively increases the QJump level for workloads that do not meet their SLOs. We add support for calculating the latency for each class as well as the end-host delay, which allows QJump++ to check if a workload can meet its SLO.

Additionally, we find that instantiating the QJump classes using the QJump equation (Equation (4) in [34]) severely limits the number of admitted workloads (5x fewer on average). By fixing a set of throughput values independent of the number of workloads, we significantly boost the number of admitted workloads for QJump++.

PriorityMeister (PM) [86]: PriorityMeister uses DNC to offer each workload a worst-case request latency guarantee based on rate limits that are automatically derived from a workload's trace. PriorityMeister automatically configures workload priorities to meet latency SLOs across both network and storage and is described in Chapter 3.

4.3.2 Traces

Our evaluation uses 180 recent traces captured in 2015 from the datacenter of a large Internet company. The traces capture cache lookup requests issued by a diverse set of Internet applications (e.g., social networks, e-commerce, web, etc.). Each trace contains a list of anonymized requests parameterized by the arrival time and object size being requested, ranging from 1 Byte to 256 KBytes with a mean of 28 KBytes. Each trace is 30 minutes long and contains 100K to 600K requests, with a mean of 320K requests. We find that these traces exhibit significant short-term burstiness, and Figure 4.9 shows that the CDF for the ratio of peak to mean request rate ranges from 2 to 6. We also perform standard statistical tests [1, 58] to verify the stationarity and mutual stochastic independence of our traces as required by SNC. For our storage experiments in Section 4.4.6, we also use a set of storage traces from Microsoft production servers [47].

4.3.3 Experimental procedure

In most of our experiments, we run up to 180 workloads that replay memcached requests from each workload's associated trace. For each experiment, workloads arrive to the system one by one in a random order with a 99.9% SLO drawn uniformly randomly from {10ms, 20ms, 50ms, 100ms}. When a workload arrives, the admission system makes its decision based on the workload's SLO and the first half of the workload's trace (15 mins). After the admission decisions for all 180 workloads have been made, each admitted workload starts a VM to replay the second half of its request trace (15 mins). All workloads replay their traces in an open loop fashion, which properly captures the end-to-end latency and the effects of end-host queueing [66]. All admission systems meet the workload SLOs, as verified by monitoring the total memcached request latency for every request (i.e., completion time - arrival time in the trace) and checking that the 99.9% latency across 3min time intervals for each workload is less than its SLO. Thus, we evaluate the performance of the admission control systems under the following two metrics:

1. the number of workloads admitted by each system
2. the total volume of bytes transmitted by admitted workloads

Metric 1 indicates how many workloads with tail latency SLOs can be concurrently supported by each system. Metric 2 prevents a system from scoring high on metric 1 by admitting only low-load workloads.

4.3.4 Experimental testbed

Our experimental testbed comprises an otherwise idle, 18-machine cluster of Dell PowerEdge 710 machines, configured with two Intel Xeon E5520 processors and 16GB of DRAM. We use the setup shown in Figure 4.1. Six machines are dedicated as memcached servers running the most recent version (1.4.25) of memcached. Twelve machines run a set of workload VMs using the standard kvm package (qemu-kvm-1.0) to provide virtualization support. Each workload VM runs 64-bit Ubuntu 13.10 and replays a trace using libmemcached. Each physical machine runs 64-bit Ubuntu 12.04, and we use the associated Linux Traffic Control interface without modifications. The top-of-rack switch connecting the machines is a Dell PowerConnect 6248 switch, providing 48 1Gbps ports, with DSCP support for 7 levels of priority.

4.4 Results

In this section, we experimentally evaluate the performance and practicality of SNC-Meister. Section 4.4.1 shows that SNC-Meister is able to support 75% more tail latency

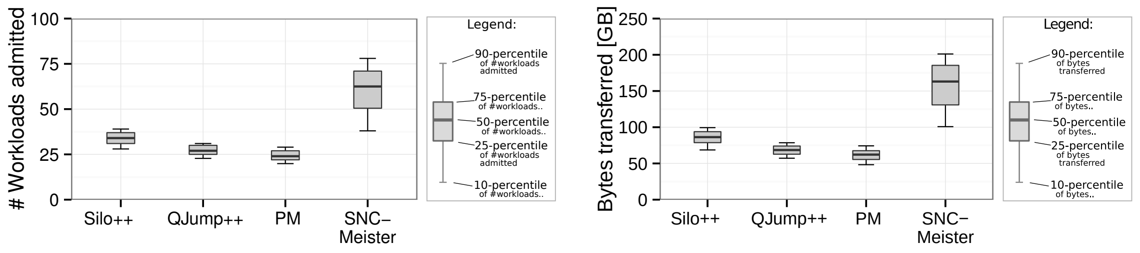


Figure 4.10: Comparison of three state-of-the-art admission control systems to SNC-Meister for 100 randomized experiments. In each experiment, 180 workloads, each submitting hundreds of thousands of requests, arrive in random order and seek a 99.9% SLO randomly drawn from $\{10\text{ms}, 20\text{ms}, 50\text{ms}, 100\text{ms}\}$. The left box plot shows that across the 100 experiments, SNC-Meister admits more workloads than state-of-the-art systems. The right plot shows that SNC-Meister achieves a similar improvement with respect to the volume of bytes transferred in each experiment.

SLO workloads than state-of-the-art systems across a large range of experiments. SNC-Meister also transfers 88% more bytes, which shows that SNC-Meister supports a higher network utilization. Section 4.4.2 shows that SNC-Meister's performance is within 7% of an empirical offline solution. Section 4.4.3 demonstrates that SNC-Meister is able to support low-bandwidth workloads with very tight SLOs alongside high-bandwidth workloads. Section 4.4.4 investigates the sensitivity of the SNC latency prediction to the SLO percentile. Section 4.4.5 evaluates the scalability of SNC-Meister and shows that both its computation time and performance scale linearly with the number of workloads. Section 4.4.6 demonstrates that SNC-Meister also extends to storage.

4.4.1 SNC-Meister outperforms the state-of-the-art

This section compares SNC-Meister with enhanced versions of the state-of-the-art tail latency SLO systems (described in Section 4.3.1). We run 100 experiments, each with 180 workloads arriving in a random order with random SLOs (described in Section 4.3.3). All four systems, including SNC-Meister, meet the SLOs for all admitted workloads, but differ in how many workloads each system admits.

Figure 4.10 shows a box plot of the number of admitted workloads and a box plot of the volume of transferred bytes. We see that the three state-of-the-art systems (Silo++, QJump++, PriorityMeister) perform roughly the same as they draw upon the same underlying DNC mathematics. SNC-Meister achieves a significant improvement over all

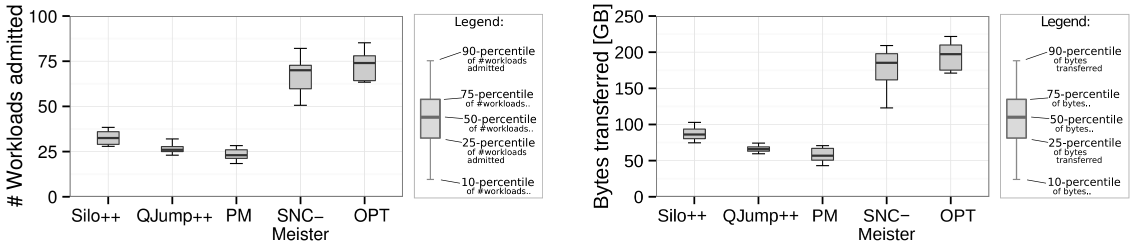


Figure 4.11: Comparison between state-of-the-art systems, SNC-Meister, and an empirical optimum (OPT) for 10 of the 100 experiments in Figure 4.10. The left box plot shows that the number of workloads admitted by SNC-Meister is close to OPT, whereas the other state-of-the-art systems admit less than half of OPT. The right plot shows that SNC-Meister is also close to OPT with respect to the volume of bytes transferred in each experiment.

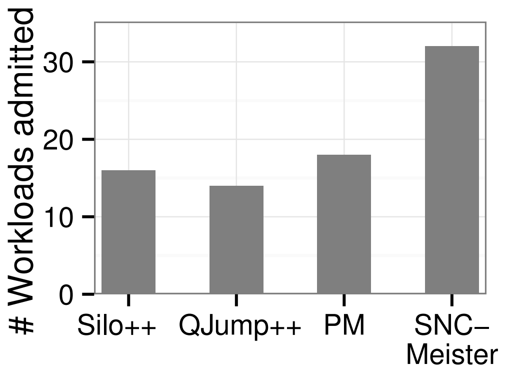
three systems across all 100 experiments. Silo++ admits slightly more than QJump++ and PriorityMeister, which is due to the effective bandwidth enhancement of Silo++ (see Section 4.3.1). Nevertheless, SNC-Meister outperforms Silo++ by a large margin: of the 100 experiments, the 10-percentile of SNC-Meister is above the 75-percentile of Silo++ for both the number of admitted workloads and bytes transferred. The fact that SNC-Meister performs well for both metrics shows that SNC-Meister's improvement is not just due to admitting more low-load workloads, but is due to allowing higher utilization.

4.4.2 Comparison to empirical optimum

To evaluate how well SNC-Meister compares to an empirical optimum, we determine the maximum number of workloads that can be admitted without SLO violations (labeled OPT) via trial and error experiments. In order to determine the maximum in a reasonable time frame, OPT only considers workloads in the order that they arrive. Thus, OPT is defined as the largest *n* such that the first *n* workloads to arrive meet their SLOs. Determining OPT via trial and error is time consuming and hence we only do this for a random subset[5] of 10 out of the 100 experiments from Section 4.4.1.

Figure 4.11 compares the state-of-the-art systems, SNC-Meister, and OPT. We find that SNC-Meister performs almost as well as OPT (within 7% of OPT in the median). By contrast, the state-of-the-art systems admit only half of OPT. Thus, SNC-Meister captures most of the statistical multiplexing benefit without resorting to trial and error experiments.

[5]Note that the results from the 10 experiments in Figure 4.11 are representative because the state-of-the-art systems and SNC-Meister perform similarly to the 100 experiments in Figure 4.10.



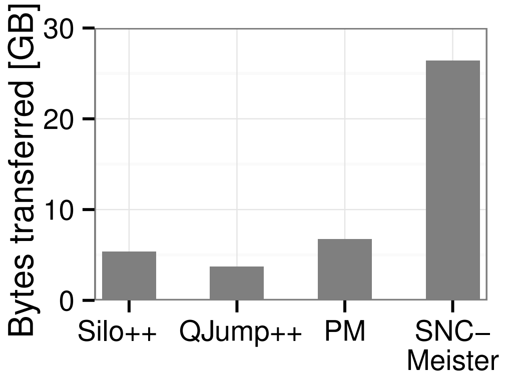


Figure 4.12: SNC-Meister's admission control performance with small-request workloads. The left graph shows the number of admitted workloads, and the right graph shows the number of bytes transferred by admitted workloads. Our experiment includes two groups of workloads: a set of small-request workloads with low-latency (4ms) SLOs and a set of large-request workloads with higher-latency (50ms) SLOs. SNC-Meister admits more workloads and over three times as many bytes as the state-of-the-art systems.

4.4.3 Small-request workloads

While we have focused on request latency, many related works focus on packet latency and the effects on small requests (i.e., single packet-sized requests). As SNC-Meister supports prioritization (Section 4.2), we demonstrate that SNC-Meister can also support workloads with small requests and very tight SLOs. Figure 4.12 shows the results from an experiment with a set of eleven workloads with single packet requests and tight SLOs (4ms) along with twenty-one other workloads with larger requests and higher SLOs (50ms). Like before, we see that SNC-Meister is able to admit many more workloads than the state-of-the-art systems. Here, PriorityMeister does better than Silo++ and QJump++ since it does not need to reserve a lot of bandwidth for the tight SLOs. Nevertheless, all three of these state-of-the-art systems suffer from the drawbacks of DNC and are unable to admit many of the large-request workloads once they've admitted the small-request workloads with tight SLOs. This can particularly be seen in the graph of the number of bytes transferred by admitted workloads. SNC-Meister admits both the small-request workloads as well as many large-request workloads, resulting in a higher network utilization. SNC-Meister is able to do so since, probabilistically, the small-request workloads do not have a big effect on the large-request workloads.

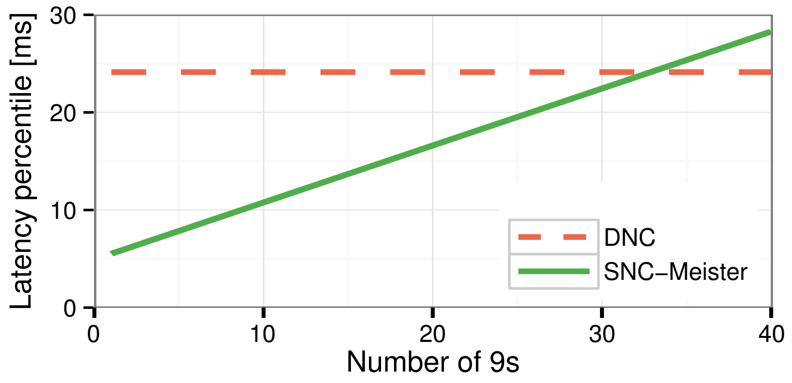


Figure 4.13: Comparison between the latency predictions of SNC-Meister and DNC for different SLO percentiles. Specifically, the x-axis denotes the number of 9s, where three 9s represents the 99.9th percentile. As expected, the latency using SNC increases with the SLO percentile, but is still superior to DNC.

4.4.4 Tail latency percentiles

One might wonder how SNC-Meister performs for latency SLOs other than the 99.9th percentile. We address this question by comparing SNC-Meister's latency prediction to the DNC latency prediction used by state-of-the-art systems.

Figure 4.13 shows the latency prediction of SNC-Meister and DNC vs. the number of 9s in the SLO percentile, where three 9s represents the 99.9th percentile. SNC-Meister's latency increases with the SLO percentile, as expected, and only exceeds the DNC latency with thirty-three 9s. Thus, SNC-Meister's benefit primarily comes from considering non-100th percentile SLOs. The relative difference between one and three 9s is small compared to the difference between non-100th percentiles (i.e., SNC-Meister) and the 100th percentile (i.e., DNC). Thus, experiments with 90th percentile SLOs and experiments where workloads have mixtures of 90th, 99th, and 99.9th percentile SLOs show similar results to the case with all workloads having 99.9th percentile SLOs. DNC's worst-case analysis is conservative in accounting for rare events that probabilistically should never occur, whereas SNC-Meister is unaffected by these improbable events for almost any percentile.

4.4.5 Scalability

In this section, we study the scalability of SNC-Meister. Figure 4.14 shows the runtime for computing latency as a function of the number of workloads. We see that SNC-Meister's

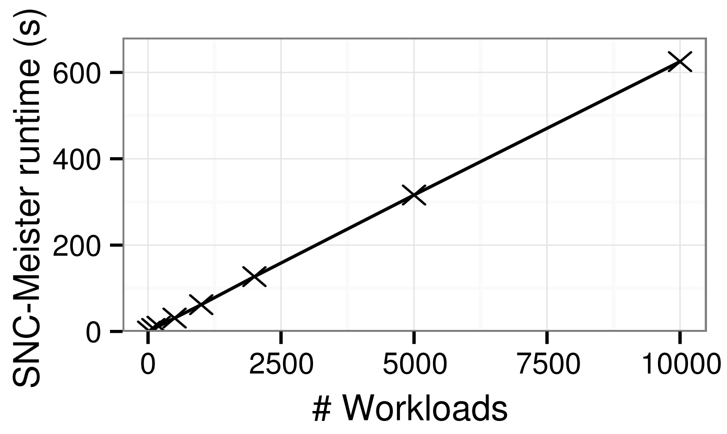


Figure 4.14: Scalability of SNC-Meister's computation. SNC-Meister's runtime scales linearly with the number of workloads.

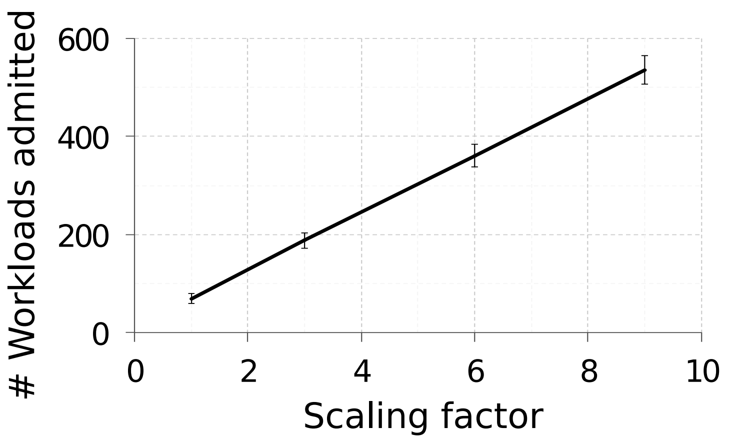


Figure 4.15: Scalability of SNC-Meister's admission. The number of admitted workloads in SNC-Meister scales linearly with the cluster size.

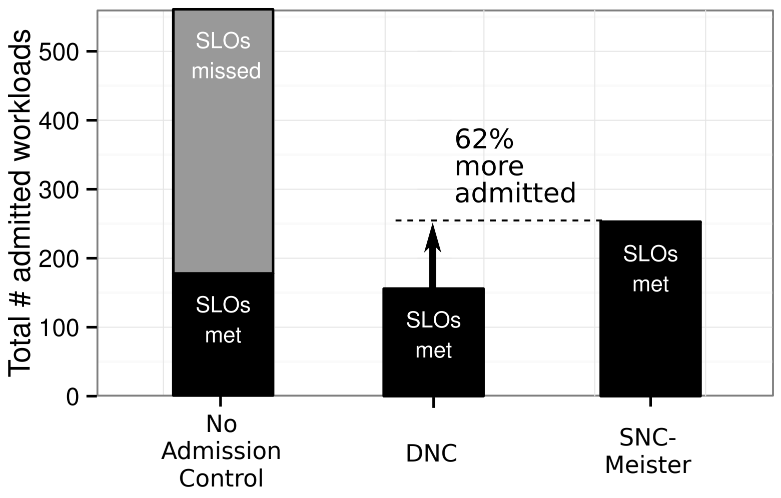


Figure 4.16: A summary of experimental results from our networked storage system involving 16 experiments, each with 35 workloads, totaling 560 workloads. The total load in each experiment is less than 60%. Without Admission Control, all 560 workloads are admitted, but 383 workloads (shaded area) violate their tail latency SLOs. With a DNC-based approach, we meet all SLOs, but only admit 156 out of 560 workloads. SNC-Meister admits 62% more workloads than DNC-based approaches.

runtime scales linearly with the number of workloads, which is ideal since each workload's latency is calculated one by one. This is promising, given that the computation is currently single threaded, and the analysis of each of the workloads can easily be parallelized.

Figure 4.15 shows how the number of admitted workloads scales with the size of the cluster. We use the same setup with 180 workloads, but replicate the workloads and number of machines by a scaling factor (x-axis) to show the effect of larger scale. The order and assignment of workload VMs to data servers is random as before. As expected, the number of workloads admitted by SNC-Meister scales linearly with the size of the cluster.

4.4.6 Storage

While our results so far focus on networks, SNC-Meister also extends to environments with storage and networks. Figure 4.16 shows a summary of our results comparing SNC-Meister, DNC, and No Admission Control. We use PriorityMeister for the DNC-based approach since it is the only DNC-based system that supports both storage and networks. We run 16 experiments, each with 35 workloads based on traces from Microsoft production servers [47]. We see that SNC-Meister is able to admit 62% more workloads than a DNC-

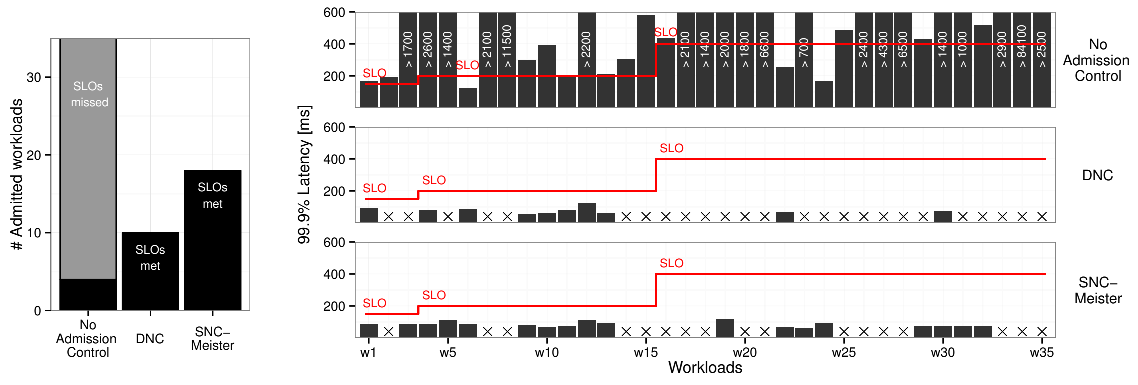


Figure 4.17: Results when running workloads with a mixture of 99.9% SLOs at 150ms, 200ms, and 400ms. The left graph shows the number of admitted workloads under the No Admission Control, DNC, and SNC-Meister policies. The right graphs show the 99.9% latency (y-axis) for each of the 35 workloads (x-axis) running on our cluster. The red solid line indicates the SLO value for each workload, where lower numbered workloads have been assigned a lower SLO. Under No Admission Control, almost all workloads exceed their SLOs. Under DNC, there are zero violations with only 29% of the workloads admitted. Under SNC-Meister, there are again zero violations with 80% more workloads admitted than under DNC.

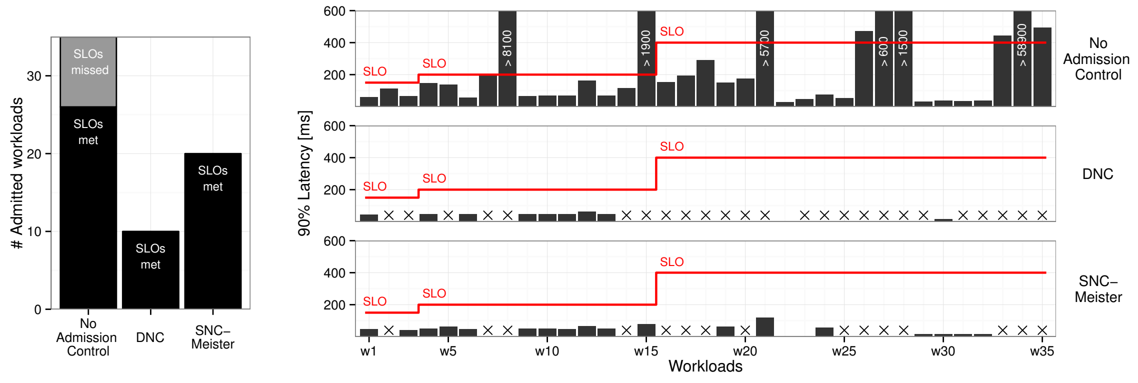
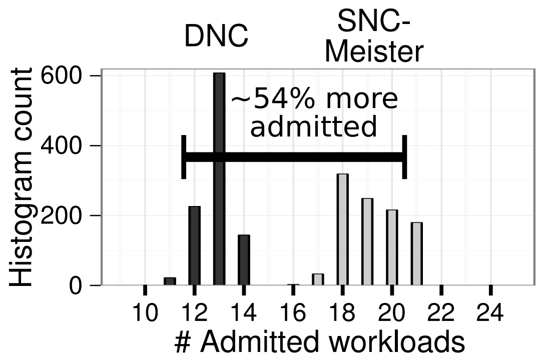
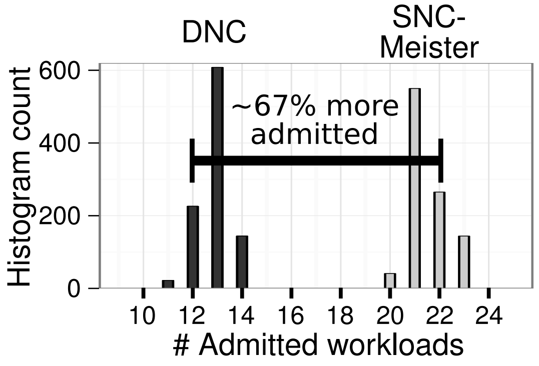


Figure 4.18: Same experiment as in Figure 4.17 except with 90% SLOs. Even for a relatively low 90th percentile, it is still possible to exceed SLOs with No Admission Control. DNC admits the exact same workloads as in Figure 4.17 since it cannot distinguish between different SLO percentiles. SNC-Meister admits twice as many workloads as DNC while still meeting SLOs.



(a) 99.9% SLOs



(b) 90% SLOs

Figure 4.19: Histogram on the number of admitted workloads when evaluating SNC-Meister and DNC on 1000 random mixtures of SLOs ranging from 150ms to 500ms. SNC-Meister admits 54% to 67% more workloads than DNC with 99.9% and 90% SLOs respectively. The clear separation between SNC-Meister and DNC indicates that SNC-Meister is admitting more workloads than DNC in virtually all cases.

based approach. No Admission Control admits all of the workloads, but 68% of them (shaded area) violate their tail latency SLOs even though the total load in each experiment is less than 60%. Thus, admission control is crucial for meeting tail latency SLOs, even when not at high load.

Figure 4.17 and Figure 4.18 show a more detailed view of two experiments at the 99.9th and 90th percentiles, respectively. The left graph shows the number of workloads admitted and the right graphs show the corresponding tail latencies of each of the workloads. The red SLO line shows the SLOs for each of the workloads, which vary in these experiments from 150ms to 400ms. In both the 99.9th and 90th percentile cases, No Admission Control misses the SLO for many workloads, with some of the violations greater than 10 times the SLO latency. DNC meets all SLOs, but only admits 29% of the workloads. Since DNC cannot distinguish between a 99.9% and a 90% SLO, it conservatively admits only 29% in both cases. By contrast, SNC-Meister admits 51% of the workloads in the case with 99.9% SLOs and 57% of the workloads in the case with 90% SLOs while also meeting all SLOs.

In Figure 4.19, we consider a broader sweep of SLOs. We generate 1000 random sets of 35 SLO latencies for each workload ranging from 150ms to 500ms. We compare SNC-Meister and DNC and find that SNC-Meister admits 54% more workloads with 99.9% SLOs (Figure 4.19(a)) and 67% more workloads with 90% SLOs (Figure 4.19(b)). Furthermore, these histograms have a clear separation between SNC-Meister and DNC, which indicates that SNC-Meister admits more workloads than DNC in almost all cases.

			tail latency SLO	multi-tenancy	parameter configuration
guaranteeing tail latency	SNC-based	SNC-Meister	Any (e.g., 99.9th)	high	automated
	worst-case admission control	Silo [43]	100th	low	manual
		QJump [34]	100th	low	manual
		PriorityMeister [86]	100th	low	automated
reducing tail latency	datacenter scheduling	pHost [29]	no	n/a	manual
		Fastpass [62]	no	n/a	manual
		pFabric [5]	no	n/a	manual
	congestion control	D2TCP [73]	no	n/a	n/a
		DCTCP [2]	no	n/a	n/a
	other	[20, 42, 70, 74, 81, 84]	no	n/a	n/a

Table 4.2: Comparison of SNC-Meister's related work. While many systems aim to reduce tail latency (bottom half of table), few provide tail latency guarantees (top half).

4.5 Related work

SNC-Meister addresses meeting tail latency SLOs, which is an active research area with a rich literature. The related work can be divided into four major lines of work. First, there is a body of work that ensures that tail latency SLOs are met based on worst-case latency bounds; unfortunately these works are unable to achieve high degrees of multi-tenancy due to the conservative nature of worst-case analysis. To overcome these limitations, theoreticians have developed a second line of work that provides probabilistic tail latency bounds via Stochastic Network Calculus (SNC). These works are entirely theoretical and have never been implemented for any computer system. Third, there is a body of work that proposes techniques for significantly reducing the tail latency; ensuring that request latency SLOs are met is not within the scope of that work. Fourth, there are some recent systems that try to meet SLOs based on measured latency; unfortunately, these works aren't suited for admission control and don't cope well with bursty workloads.

Guaranteed latency systems

There are three recent state-of-the-art systems that provide SLO guarantees: Silo [43], QJump [34], and PriorityMeister [86] (described in Section 4.3.1 and listed in the top half of Table 4.2). All three systems are designed for worst-case latency guarantees. For workloads seeking a lower percentile tail guarantee (e.g., a guarantee on the 99.9th percentile of latency), these systems are overly conservative in their admission decisions (see Section 4.4.1).

Besides these recent proposals, there has been a long history of DNC-based worst-case latency admission control algorithms in the context of Internet QoS [24, 49, 53, 72, 78]. These older proposals are not tailored to datacenter applications, and also suffer from the conservative nature of worst-case latency guarantees.

Stochastic Network Calculus (SNC)

The modern SNC theory evolved as an alternative to the DNC theory to capture statistical multiplexing gains and enable accurate guarantees for any latency percentile [10, 12, 13, 14, 16, 17, 19, 25, 27, 28, 30, 48, 54, 63, 64, 69, 82]. However, all of this work is in theory, and we are not aware of any implementations that use SNC in computer systems. The only practical applications of SNC are in the modeling of critical infrastructures such as avionic networks [65] and the power grid [31, 77], which support the robustness of SNC theory.

Reducing tail latencies

There are many systems that demonstrate how to reduce tail latency (listed in the bottom half of Table 4.2). Datacenter schedulers, such as pHost [29], Fastpass [62], and pFabric [5], improve tail latency by bringing near-optimal schedulers (such as earliest-deadline first) to the datacenter. These approaches can also shift queueing from within the network to the end-hosts, which greatly reduces tail packet latency and the latency of short requests. These approaches, however, are not designed to ensure tail latency SLO compliance.

Latency-aware congestion control algorithms, such as D2TCP [73] and DCTCP [2], aggressively scale down sending rates and prioritize flows with deadlines. These approaches react to congestion, which can lead to SLO violations in the face of bursty traffic [5, 43]. HULL [4] keeps tail latencies low by controlling the network utilization through rate limiting, but can still experience SLO violations [34].

Other techniques for reducing tail latency include issuing redundant requests [20, 42, 74], latency-adaptive machine selection [70, 81], and latency-adaptive load balancing [84]. While these techniques can reduce the tail latency, they are not designed for meeting tail latency SLOs.

Measurement-based approaches

Several recent works measure the latency and adapt the system to try to meet tail latency SLOs [52, 76]. Unfortunately, these approaches aren't suited for admission control where admission decisions cannot be made dynamically. Furthermore, prior work has shown that reactive approaches struggle with bursty workloads and often do not meet their SLOs [86].

The recent Cerebro [44] work uses measurements to characterize the latency of requests composed of multiple sub-requests. Unlike SNC-Meister, Cerebro is not designed to account for the interaction between multiple workloads, which is a primary benefit of SNC.

4.6 Chapter summary

This chapter investigates how to perform admission control with various tail latency percentiles. Admission control is a critical feature for ensuring good performance for admitted workloads. However, admission control for tail latency is particularly difficult since tail latency is hard to analyze, especially with bursty workloads. If latency estimates are too conservative, then the admission controller will not be able to admit many workloads.

SNC-Meister is a new admission control system for meeting tail latency SLOs while achieving higher multi-tenancy than the state-of-the-art. In experiments with production traces on a physical implementation testbed, we show that SNC-Meister can admit two to three times as many workloads as the state-of-the-art while meeting tail latency SLOs. SNC-Meister benefits from applying a new probabilistic theory called Stochastic Network Calculus (SNC) to calculate tail latencies, while prior systems use the conservative worst-case Deterministic Network Calculus (DNC) theory. In fact, SNC-Meister is the first computer system to practically use SNC.

As SNC is a new theory, there are many challenges in bringing it to practice, and there is much room for further research. One challenge we identify is the important role of the order in which SNC operators are applied – a fundamental problem that was not previously considered in SNC literature. Our novel algorithm for analyzing networks with SNC makes a significant step forward in making SNC a practical tool. We also add support in SNC-Meister for dependencies between subsets of workloads, which addresses a practical issue that is generally ignored in SNC theory. Our SNC library is now publicly open-sourced at `https://github.com/timmyzhu/SNC-Meister`.

While this work focuses on the admission control problem, the ideas behind SNC-Meister and SNC are applicable to many settings beyond admission control. One such example is the datacenter provisioning problem. By being able to analyze workload behavior and compute tail latency, SNC-Meister could be extended to deciding if (and how many) more resources are required for meeting tail latency SLOs. Similarly, these techniques could apply to workload placement problems. SNC could be used to identify bottlenecks and make placement decisions in a tail latency aware fashion. We thus believe that the SNC theory can develop into a practical tool for working with tail latency.

While SNC is a great tool, it is not suited for every scenario. In settings where workloads may be adversarial or correlated in unknown ways, DNC is more applicable than SNC. DNC is also necessary for real-time settings where 100th percentile guarantees are desired. So it is worthwhile building systems based on both SNC and DNC, and in the next chapter, we'll introduce WorkloadCompactor, a DNC-based workload placement system.

Chapter 5

WorkloadCompactor: Reducing datacenter cost while providing tail latency SLO guarantees

Service providers want to reduce datacenter costs by consolidating workloads onto fewer servers. At the same time, customers have performance goals, such as meeting tail latency SLOs. In this chapter, we answer the question of how to consolidate networked storage workloads onto storage servers while meeting tail latency SLOs.

To limit interference when consolidating workloads, customers and service providers often agree upon rate limits. Ideally, rate limits are chosen to maximize the number of workloads that can be co-located while meeting each workload's SLO. In reality, neither the service provider nor customer knows how to choose rate limits. Customers end up selecting rate limits on their own in some ad hoc fashion, and service providers are left to optimize given the chosen rate limits.

We present WorkloadCompactor, a new system for automatically choosing rate limits – simultaneously with selecting on which server to place workloads – to minimize cost while meeting tail latency SLOs. A key finding in our work is that the ability to co-locate workloads is significantly impacted by how rate limits are chosen. In fact, the optimal choice of a rate limit depends on how rate limits are selected for other co-located workloads. WorkloadCompactor introduces:

1. a novel approach for jointly optimizing a set of workloads' rate limits
2. a scalable placement algorithm for placing workloads onto servers

Our experiments show that by optimizing the choice of rate limits, WorkloadCompactor reduces the number of required servers by 30-60% as compared to state-of-the-art approaches while meeting tail latency SLOs.

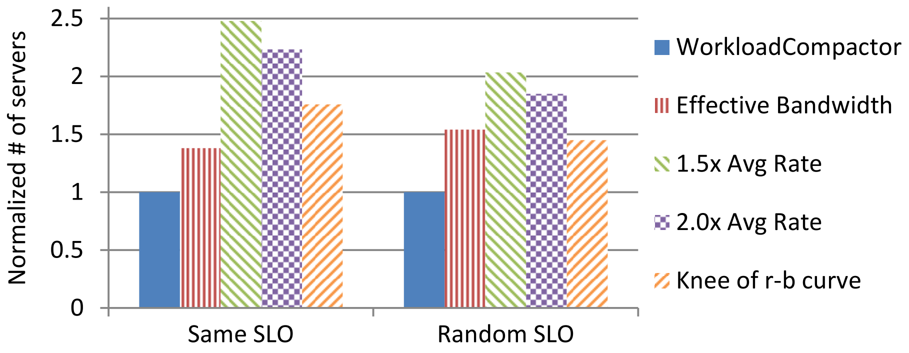


Figure 5.1: Comparing WorkloadCompactor to state-of-the-art approaches under two scenarios, each using 1000 workloads based on production traces. In the first scenario, all workloads specify the same SLO, and in the second scenario, workloads specify random SLOs. Results are normalized to the number of servers used by WorkloadCompactor to clearly show that state-of-the-art approaches require 40-150% more servers than WorkloadCompactor.

We introduce the problem and discuss the scope of this chapter in Section 5.1. We present the design and implementation of WorkloadCompactor in Section 5.2. We then describe our experimental setup in Section 5.3 followed by our results in Section 5.4. We discuss related work in Section 5.5 and conclude with a summary of this chapter in Section 5.6.

5.1 Introduction

In cloud computing and enterprise datacenter environments, service providers often seek to maximize the utilization of their resources by *sharing* compute, network, and storage resources among customers. At the same time, service providers want to keep their customers happy by providing good performance. Some customers may specify their performance goals in terms of a *tail latency* Service Level Objective (SLO), such as “99% of requests must complete within 150 milliseconds”. Cloud researchers and companies such as Amazon and Google have repeatedly stressed the importance of meeting tail latency SLOs at, for example, the 99th and 99.9th percentiles, particularly for user-facing interactive applications [5, 20, 21, 34, 43, 62, 73, 81, 86, 87].

Our goal is to simultaneously achieve these two objectives: (1) ensure that all workload tail latency SLOs set by customers are met, while (2) minimizing the number of resources

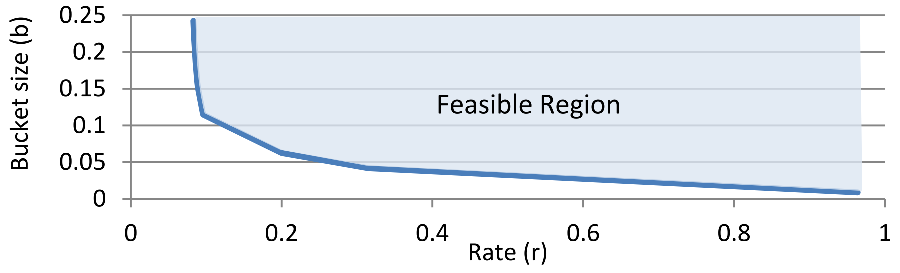


Figure 5.2: Characterizing burstiness via an r-b tradeoff curve. Feasible $\langle r, b\rangle$ points represent rate limit parameters such that a workload is not delayed by the rate limiter.

that the service provider must devote to satisfying those workloads.

Our work targets network attached storage, such as Amazon's Elastic Block Store (EBS). Specifically, we consider the problem of how to share networked storage, which involves deciding where to place storage workloads and how much resource to allocate to each workload. The current practice for allocating storage resources is to have customers either reserve some amount of storage throughput (e.g., Amazon's Provisioned IOPS) or run without any guarantees in a best effort fashion. Unfortunately, reserving throughput is inefficient since customers end up reserving more throughput than necessary to handle bursty behavior. In research, multiple papers have proposed using token bucket rate limits with both a rate (i.e., throughput) parameter (r) and a burst (a.k.a. bucket size) parameter (b). For example, Silo [43] and QJump [34] successfully use rate limiting to provide network latency guarantees, and Avatar [85] and pClock [35] similarly manage storage latency via rate limiting. However, an open problem in all of these papers is a method for how to *choose* the rate limit parameters. They all assume the customer provides the rate limit parameters as input.

Our results show that selecting rate limit parameters is an important problem that can result in using 2.5x more servers than necessary (Figure 5.1). The reason for such a big difference is because there are (infinitely) many feasible $\langle r, b\rangle$ choices for a given workload, and the choice of an $\langle r, b\rangle$ tuple affects how easily it can be co-located with other workloads. We define *feasible* $\langle r, b\rangle$ tuples for a workload as rate limit parameters such that the rate limiter is sufficiently large enough to process workload requests without delay. Figure 5.2 shows an example of feasible $\langle r, b\rangle$ tuples where all points on or above the r-b curve are feasible.

There are multiple approaches to choosing an $\langle r, b\rangle$ tuple. Authors of the recent Silo [43] paper, for example, select rate limits in their experiments by setting r to the average rate multiplied by some constant k (e.g., $k = 1.5$). The bucket size b can then be set by trial and error experiments or via the *r-b* curve. We refer to this approach as 1.5x Avg Rate or 2.0x Avg Rate. Another natural heuristic, which we call "Knee of *r-b* curve", is to select the knee of the *r-b* curve in hopes of making a good tradeoff between *r* and *b*. The state-of-the-art in theory for selecting rate limits is based on the Effective Bandwidth approach from deterministic network calculus [51]. Unfortunately, all of these approaches are quite suboptimal in minimizing the number of servers. This is because there is no "best" $\langle r, b\rangle$ tuple for a workload; the optimal $\langle r, b\rangle$ tuple depends on the other workloads sharing the server.

In fact, to minimize the number of required servers, a solution needs to be able to dynamically reselect $\langle r, b\rangle$ tuples for existing workloads as new workloads arrive. This introduces many challenges. First, the solution needs to predict how latency is affected by the choice of $\langle r, b\rangle$ tuples for workloads sharing a server. Measurement based approaches are insufficient for considering all of the many $\langle r, b\rangle$ combinations across the workloads sharing a server; an analytic approach such as deterministic network calculus is necessary.

Second, latency is affected by various stages of a request. For networked storage, a request traverses the network to the server, accesses the storage device on the server, and traverses the network back to the client. Each of these stages has different requirements based on the types of requests a workload sends. For example, the network traffic leaving the server would send the number of bytes accessed for read requests and a constant (i.e., size of acknowledgment) for write requests. Each of these stages needs to be represented by its own *r-b* curve, and the solution needs to pick $\langle r, b\rangle$ tuples for each of these stages for each of the workloads sharing a server.

In addition to these challenges, there is also the problem of deciding onto which server to place a workload. When placing a new workload, considering each server in a first-fit fashion is slow, and a more scalable approach is needed.

WorkloadCompactor is a new system that solves all of these challenges. It includes a tool for generating *r-b* curves based on traces of workload behavior (see Section 2.3.1). Given *r-b* curves for each workload, it *automatically* chooses both storage and network rate limit parameters for each workload – simultaneously with selecting onto which server to place storage workloads. The *key novelty* to WorkloadCompactor's workload compaction algorithm is a *specially formed linear program (LP) based on equations from deterministic network calculus*. The LP optimizes the choice of $\langle r, b\rangle$ tuples across all stages and across all workloads sharing a server. WorkloadCompactor also provides a scalable heuristic for how to quickly decide onto which server to place workloads.

Figure 5.1 shows that WorkloadCompactor's ability to dynamically reconfigure workload rate limits provides a significant advantage over state-of-the-art heuristics in compacting workloads while satisfying tail latency SLOs. Our experiments with production workload traces show that while all approaches meet tail latency SLOs, the state-of-the-art approaches do so using 40-150% more servers than WorkloadCompactor.

In this chapter, we make the following main contributions:

- **Building an automated system for minimizing the number of servers to meet tail latency SLOs:** WorkloadCompactor is a new QoS system that enforces rate limits and priorities in storage and network to meet tail latency SLOs. WorkloadCompactor minimizes the number of servers using a new technique for automatically selecting rate limits and priorities to compact more workloads onto a server while meeting SLOs; our technique is based on non-trivial applications of deterministic network calculus. Our compaction technique is used in conjunction with our scalable placement algorithm, which places workloads onto servers an order of magnitude faster than the traditional first-fit policy.
- **Extensive evaluation:** We evaluate WorkloadCompactor on a physical 24-machine cluster using 62-85 workloads derived from real production traces to demonstrate that WorkloadCompactor uses 30-60% fewer servers than state-of-the-art approaches while meeting tail latency SLOs. Our scalability experiments with 1000 workloads show that WorkloadCompactor is able to quickly and effectively pack workloads at large scale. We also show that WorkloadCompactor works well in a broad range of scenarios such as mixing workload arrivals/departures and using multiple SSDs per server.

5.2 WorkloadCompactor

We target storage workloads, which send a stream of requests (e.g., read, write) over the network to access data on storage servers. We imagine Amazon's Elastic Block Store (EBS) as a typical example scenario for WorkloadCompactor, but the techniques described are applicable to other systems such as NFS servers, memcached servers, or databases. In this example scenario, an Amazon customer runs a workload (e.g., mail server) on an "instance"[1] connected to one or more "EBS volumes". An EBS volume is hosted on a storage server, which provides networked storage to the volume's connected instance. WorkloadCompactor is responsible for helping the service provider, Amazon in this case, decide onto which storage server to host an EBS volume along with rate limits for workloads accessing the storage server.

Figure 5.3 shows the process of adding a workload in WorkloadCompactor. When a customer wishes to add a workload (e.g., a database backed by an EBS volume), the customer allocates an instance for the database in the usual way. However, when the

[1]Instances represent virtual machines (VMs) in Amazon, but the design of WorkloadCompactor does not require virtualization for either the client or server.

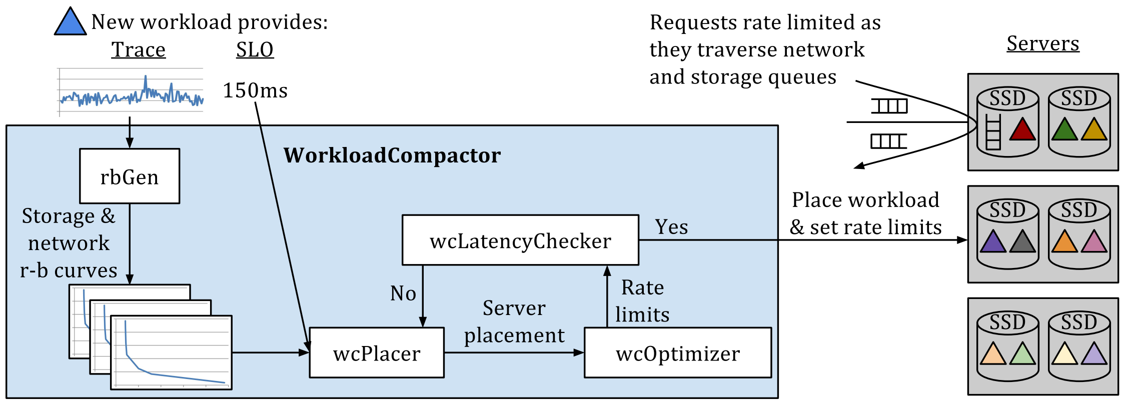


Figure 5.3: WorkloadCompactor system diagram.

customer allocates the EBS volume, the customer specifies the desired latency along with a description of the workload's storage and network utilization in the form of a historic trace. The trace gets translated into *r-b* curves via our `rbGen` tool (Section 2.3.1). The customer can also specify a safety margin to scale the *r-b* curves to account for deviations in past behavior; we explore the robustness of *r-b* curves in Section 5.4.2. The provider then provides the desired level of service as specified by the SLO and *r-b* curves.

Having the *r-b* curve rather than a single $\langle r, b \rangle$ point is important since the choice of a specific $\langle r, b \rangle$ point has a significant impact on the ability to co-locate workloads. If all workloads select rate limits with low rates and large bucket sizes, then it will be hard to co-locate workloads since the large bucket sizes will allow large bursts that could violate SLOs. If all workloads select rate limits with high rates and small bucket sizes, then it will be hard to co-locate workloads since all of the available bandwidth will quickly be used up.

Given the tail latency SLO and *r-b* curves for a workload, WorkloadCompactor decides onto which server to place the workload, along with what rate limits to set for the workload. First, the *wcPlacer* component identifies candidate servers upon which to place the workload. Second, the *wcOptimizer* component speculatively determines candidate $\langle r, b \rangle$ tuples for each workload on the server. Third, the *wcLatencyChecker* component determines whether the candidate placement and $\langle r, b \rangle$ tuples would satisfy all workload SLOs. If not, the cycle begins again with the wcPlacer identifying a new candidate server. Instead, if all SLOs are satisfied, WorkloadCompactor configures the appropriate storage and network rate limits and completes by assigning the workload to the server.

5.2.1 *wcLatencyChecker*: Guaranteeing SLOs

WorkloadCompactor relies upon deterministic network calculus, which has been shown to be effective in related literature [34, 43, 86]. Deterministic network calculus provides a framework for calculating latency guarantees based on the selected $\langle r, b \rangle$ tuples. Specifically, we use deterministic network calculus equations to compute the latency due to queueing at a server; we write equations from the perspective of a single server and repeatedly apply them to each server.

For simplicity of exposition, we start by showing how to handle a single stage (e.g., storage), and later show how to extend to multiple stages. For *any* workload at priority p, the following equation calculates an upper bound on tail latency:

$$latency(p) \leq \frac{\sum\limits_{j|p_j \geq p} b_j}{1 - \sum\limits_{j|p_j > p} r_j} \tag{5.1}$$

where $\langle r_j, b_j \rangle$ corresponds to workload j's selected rate limit. The numerator is the sum of bucket sizes b_j across workloads j where j's priority, denoted by p_j, is higher than or equal to p. The denominator is 1 minus the sum of rates r_j across workloads j where j's priority p_j is strictly higher than p. Note that $\langle r_j, b_j \rangle$ is normalized such that r_j is between 0 and 1.

From Equation (5.1), note that prioritization provides the benefit that workloads are only affected by equal or higher priority workloads. WorkloadCompactor uses prioritization to provide better latency for the workloads with tighter SLO constraints. Specifically, WorkloadCompactor sets priorities in order of SLOs such that workloads with tighter SLOs are assigned higher priorities. In other words, each priority p is associated with an SLO, denoted by SLO_p, where $p1 > p2$ implies $SLO_{p1} < SLO_{p2}$.

5.2.2 *wcOptimizer*: Selecting optimal rate limits

The choice of $\langle r, b \rangle$ parameters has a significant impact on how many workloads can be co-located onto servers. Rather than using ad hoc approaches to choose the rate limit parameters, WorkloadCompactor introduces a novel systematic approach for optimizing the $\langle r, b \rangle$ parameters; existing strategies are described in Section 5.3.1. Our approach is based on two key ideas.

First, since WorkloadCompactor accepts *r*-*b* curves as input, it is able to dynamically re-select rate limit parameters. When a new workload is added to a server, Workload-Compactor recomputes rate limits for each of the workloads sharing that server. Thus, WorkloadCompactor does not need to consider future workload arrivals and only needs to optimize based on the current workloads in the system.

Second, WorkloadCompactor directly embeds Equation (5.1) into its optimization. Since Equation (5.1) is used to check if workloads can be co-located, WorkloadCompactor can check if there exists any set of rate limit parameters for the workloads such that they all can be co-located. While checking all possible rate limits may sound slow and intractable, a key insight is that we can actually represent the problem as a linear program (LP), which can be efficiently solved. Specifically, for each priority level p with a given SLO, SLO_p, we want to ensure that:

$$\frac{\sum\limits_{j|p_j \geq p} b_j}{1 - \sum\limits_{j|p_j > p} r_j} \leq SLO_p \tag{5.2}$$

which can be rewritten as the linear inequality:

$$\sum_{j|p_j \geq p} b_j + \sum_{j|p_j > p} r_j \cdot SLO_p \leq SLO_p \tag{5.3}$$

Thus, WorkloadCompactor creates an LP with r_j and b_j as LP variables representing workload j's selected rate limit $\langle r_j, b_j \rangle$. Equation (5.3) is added as a constraint for each priority level to ensure SLOs are guaranteed. Additionally, constraints are added to ensure that each selected rate limit $\langle r_j, b_j \rangle$ is on (or above) the workload's *r-b* curve. Since the *r-b* curves are piecewise linear convex functions, they can be encoded as linear constraints in the LP by taking each of the lines defined by the piecewise segments in the *r-b* curve and adding an LP constraint that $\langle r_j, b_j \rangle$ is above the line. Lastly, the following LP constraint is added to ensure the server is not overloaded:

$$\sum_j r_j \leq 1 \tag{5.4}$$

Note that the sums in these LP constraints are in the context of one specific server (i.e., the server where the new workload is being added).

WorkloadCompactor then uses an off-the-shelf solver (e.g., GLPK) to determine if the LP is feasible (i.e., there exist valid $\langle r_j, b_j \rangle$ rate limits that satisfy the constraints) or if there are no such rate limit configurations that can satisfy all workload SLOs. Since LP feasibility is the primary concern, the specific choice of objective function is not critical, and WorkloadCompactor simply minimizes the sum of rates.

To handle multiple stages (e.g., network, storage), WorkloadCompactor uses Equation (5.1) three times to represent the three stages: network into server, storage, network out of server. This results in the following equation:

$$\frac{\sum\limits_{j|p_j \geq p} b_j^{netIn}}{1 - \sum\limits_{j|p_j > p} r_j^{netIn}} + \frac{\sum\limits_{j|p_j \geq p} b_j^{storage}}{1 - \sum\limits_{j|p_j > p} r_j^{storage}} + \frac{\sum\limits_{j|p_j \geq p} b_j^{netOut}}{1 - \sum\limits_{j|p_j > p} r_j^{netOut}} \leq SLO_p \tag{5.5}$$

Unfortunately, Equation (5.5) is not a linear inequality, which makes the optimization difficult. The key trick we discovered in solving this problem is to apply a relaxation to the problem to convert it into a linear inequality. Specifically, we add a new LP variable R_p for each priority level such that it obeys the following three constraints:

$$\sum_{j|p_j > p} r_j^{netIn} \leq R_p, \quad \sum_{j|p_j > p} r_j^{storage} \leq R_p, \quad \sum_{j|p_j > p} r_j^{netOut} \leq R_p$$

Intuitively, the R_p variable balances the rate across the three stages. Equation (5.5) can then be relaxed to the inequality:

$$\frac{\sum\limits_{j|p_j \geq p} b_j^{netIn}}{1 - R_p} + \frac{\sum\limits_{j|p_j \geq p} b_j^{storage}}{1 - R_p} + \frac{\sum\limits_{j|p_j \geq p} b_j^{netOut}}{1 - R_p} \leq SLO_p \tag{5.6}$$

which can be rewritten as the linear inequality:

$$\sum_{j|p_j \geq p} b_j^{netIn} + \sum_{j|p_j \geq p} b_j^{storage} + \sum_{j|p_j \geq p} b_j^{netOut} + R_p \cdot SLO_p \leq SLO_p \tag{5.7}$$

Thus, to handle multiple stages, WorkloadCompactor replaces Equation (5.3) with Equation (5.7), and for each priority level p, it adds the R_p variable along with R_p's 3 constraints.

5.2.3 *wcPlacer*: Selecting workload placements

Since storage workloads are difficult to migrate, we restrict our design space to solutions that do not rely upon constantly migrating workloads to fix bad placements. So to make a good placement where SLOs are met, WorkloadCompactor places workloads onto servers where they fit, as determined by solving the LP (Section 5.2.2). It remains to establish the order in which to check servers for fit. Our tests with placement heuristics[2] indicate that first-fit

[2] We have tried various heuristics including first-fit, balancing the number of workloads per server, balancing the average load per server, spreading bursty workloads onto different servers, spreading workloads with different SLOs onto different servers, and random first-fit. We did not see any heuristic perform significantly better than the others, and first-fit was one of the best policies we tried.

yields good packings, which agrees with theoretical results[3]; hence, WorkloadCompactor adopts a first-fit strategy.

Unfortunately, a naïve implementation of first-fit is slow and unscalable. Often times, most servers are nearly full, so a lot of time is wasted in determining that the new workload cannot fit on near-full servers. WorkloadCompactor adds an optional fast-first-fit (FFF) placement feature where it tracks how full servers are and skips trying to place workloads onto near-full servers. Specifically, WorkloadCompactor tracks the sum of configured rates at each server and skips placing workloads onto servers where the new workload would overload the server (i.e., violate Equation (5.4)) assuming that rate limits are not reconfigured. This avoids running the LP to reconfigure rate limits, but may result in using extra servers in cases where reconfiguring rate limits would have allowed the new workload to be packed together. Our experiments (Section 5.4.3) show that FFF drastically improves the speed and scalability of WorkloadCompactor (e.g., over 10x faster with 1000 workloads) without significantly increasing the number of servers (within 3-4%).

5.3 Experimental setup

This section describes the comparison approaches, production traces, and testbed used for the performance evaluation of WorkloadCompactor.

5.3.1 Comparison approaches

To evaluate the effectiveness of WorkloadCompactor, we compare its performance to three state-of-the-art approaches to selecting a workload's rate limits: scaling average bandwidth, effective bandwidth, and finding the knee of the *r*-*b* curve. To make a fair comparison, all approaches provide tail latency SLO guarantees by adhering to Equation (5.1). Workloads are placed using a first-fit strategy, which works well, as noted in Section 5.2.3.

Scaling average bandwidth

Little is known about selecting rate limits, and most users resort to ad hoc heuristics. Authors of the recent Silo [43] paper, for example, select rate limits by setting r to the average rate of the workload multiplied by some constant k (e.g., $k = 1.5$). The b parameter can then be determined through trial and error experiments or via the *r*-*b* curve (Section 2.3.1). By

[3] Packing workloads with rate limits and priorities onto servers can be translated into the "online vector bin packing" problem where rate limits correspond to packed-object sizes and the number of priorities is correlated with the dimension of the vector. A recent STOC paper [9] proves a lower bound that is close to the known upper bound for first-fit, indicating that first-fit is near-optimal.

choosing higher *r* values, smaller bursts are allowed into the system, which allows more workloads to be co-located without violating SLOs. However, higher *r* values may also exhaust the available bandwidth. Our results evaluate this approach with two values of *k*: 1.5 and 2, corresponding to values used in Silo. We also test a range of values from 1.25 to 20, but find that all of them perform worse than the effective bandwidth approach, described next.

Effective bandwidth

The state-of-the-art in selecting rate limits is based on the effective bandwidth theory [51]. The effective bandwidth approach is designed to isolate each workload's burstiness from the other workloads in the system. Intuitively, the effective bandwidth approach slows down traffic at the rate limiter to create smooth traffic and eliminate burstiness within the system. Thus, the effective bandwidth approach sets *b* to 0 to create smooth traffic and calculates the minimum *r* (known as the effective bandwidth) such that the workload is slowed down by no more than the SLO.

The main downside to the effective bandwidth approach is that it isolates each workload's burstiness, which eliminates any multiplexing benefit in the system. Specifically, since congestion is eliminated from the system, prioritization does not provide any multiplexing benefit. Thus, the effective bandwidth approach is suboptimal in cases where prioritization is useful (i.e., workloads with different SLOs), but is reasonable in cases where prioritization is less helpful (i.e., workloads with same SLOs).

Knee of *r*-*b* curve

Looking at the shape of the *r*-*b* curves, one might consider a heuristic for selecting rate limit parameters based on the "knee" of the curve. We are not aware of any system that uses this approach, but it seems to be a reasonable way to trade off *r* and *b*. We evaluate this approach with the knee defined as the point along the *r*-*b* curve that minimizes $r+b$.

5.3.2 Traces

Our evaluation uses a collection of real production storage traces of Microsoft services (e.g., LiveMaps, Exchange), which are described in detail in [47]. In our experiments, we consider each trace to represent a workload. Half of the trace is used for generating *r*-*b* curves (Section 2.3.1), and the other half is replayed on our cluster to demonstrate that WorkloadCompactor is able to meet tail latency SLOs. We replay traces in an open loop fashion, which properly captures the end-to-end latency and the effects of queueing.

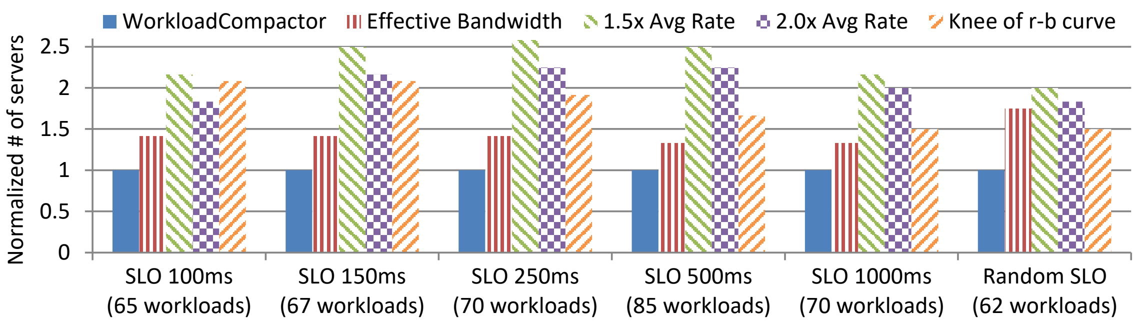


Figure 5.4: Number of servers required by state-of-the-art approaches to meet tail latency SLOs, normalized to the number of servers used by WorkloadCompactor. In all experiments, we randomly select workloads. In the first 5 "Same SLO" experiments, we use a fixed SLO for all workloads. In the last "Random SLO" experiment, workloads are configured with random SLOs from $\{$100ms, 150ms, 250ms, 500ms, 1000ms$\}$. Each of these experiments is run on our local cluster, and WorkloadCompactor is able to meet all workload SLOs while using significantly fewer servers.

5.3.3 Experimental testbed

All experimental results are run on a dedicated rack of servers, each configured with two Intel Xeon E5-2680 processors, 64GB of DRAM, and an Intel 710 series 300GB SSD. The servers are connected via a 1Gbps network. We replay traces in VMs running 64-bit Ubuntu 14.04 and use the standard NFSv3 server and client that come with these operating systems to provide remote storage access.

5.4 Results

5.4.1 WorkloadCompactor uses fewer servers

One of the surprising results in our work is that the ability to compact workloads onto servers while meeting tail latency SLOs is highly influenced by how rate limits are chosen for each workload. Figure 5.4 compares WorkloadCompactor with the state-of-the-art approaches in choosing rate limits across several experiments. In each experiment, we assign 99.9% tail latency SLOs to randomly selected workloads and count the number of servers used, normalized to the number of servers used by WorkloadCompactor. In the first 5 "Same SLO" experiments, we use a fixed SLO for all workloads. In the last "Random

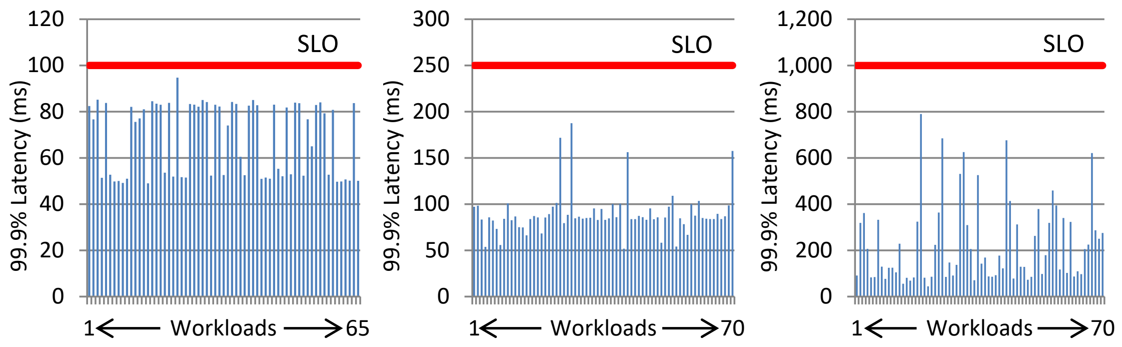


Figure 5.5: 99.9% latency (vertical bars) from running the "Same SLO" experiments in Figure 5.4 on our cluster using WorkloadCompactor. All workload 99.9% latencies are below the SLO (horizontal line).

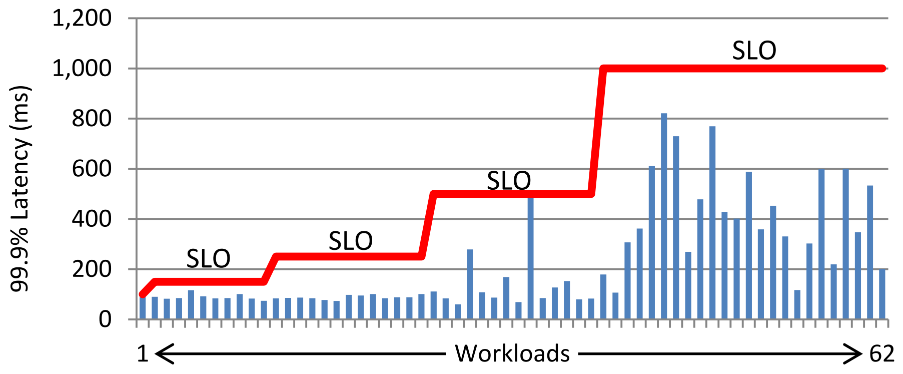


Figure 5.6: 99.9% latency from the "Random SLO" experiment in Figure 5.4 with workloads grouped by SLO.

SLO" experiment, we assign random SLOs from {100ms, 150ms, 250ms, 500ms, 1000ms}. When selecting workloads, we only consider workloads that can meet their SLOs when run in isolation to avoid using an SLO that is too tight for a workload. As a result, in experiments with higher SLOs, we randomly select from a larger pool of workloads that includes more bursty workloads.

Figure 5.4 shows that WorkloadCompactor uses far fewer servers than the state-of-the-art approaches. For the Same SLO experiments, effective bandwidth works better than the other state-of-the-art approaches, but still uses 40% more servers than WorkloadCompactor. For the Random SLO experiment, the knee method works better than effective bandwidth since the effective bandwidth approach is fundamentally unable to take advantage of

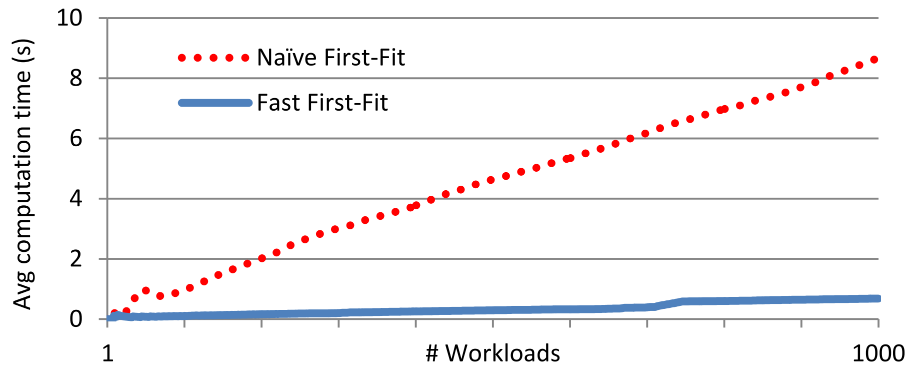


Figure 5.7: Comparing the computation time scalability of first-fit and WorkloadCompactor's fast first-fit (FFF) algorithm. FFF is much faster since it skips checking servers that are nearly full.

prioritization benefits. Nevertheless, the knee method still uses 50% more servers than WorkloadCompactor. WorkloadCompactor is the only method that works well in all cases.

5.4.2 Robustness

To demonstrate that WorkloadCompactor meets 99.9% tail latency SLOs, we measure each workload's 99.9% latency when running the experiments in Section 5.4.1 on our local cluster. Our initial results (not shown) reveal that WorkloadCompactor meets all workload SLOs when workloads are represented by their *r*-*b* curves. To explore the effect when workloads deviate from their expected behavior, we run another set of experiments where we use the first half of each workload's trace to generate *r*-*b* curves and replay the second half. We find that almost all workloads still meet their SLOs, but a few miss their SLOs due to specifying *r*-*b* curves that are too small. One way of addressing this issue is to add a "safety margin" by increasing the *r*-*b* curves. Figure 5.5 and Figure 5.6 show our experimental results with a 10% safety margin (i.e., scaling the *r*-*b* curves by 1.1); all of the workload 99.9% latencies (vertical bars) are under the SLO (horizontal line) in all experiments.

5.4.3 Scalability of computation

Figure 5.7 shows the scalability of WorkloadCompactor's *computation* as the cluster size grows. Our results show that WorkloadCompactor's fast first-fit (FFF) policy (Section 5.2.3)

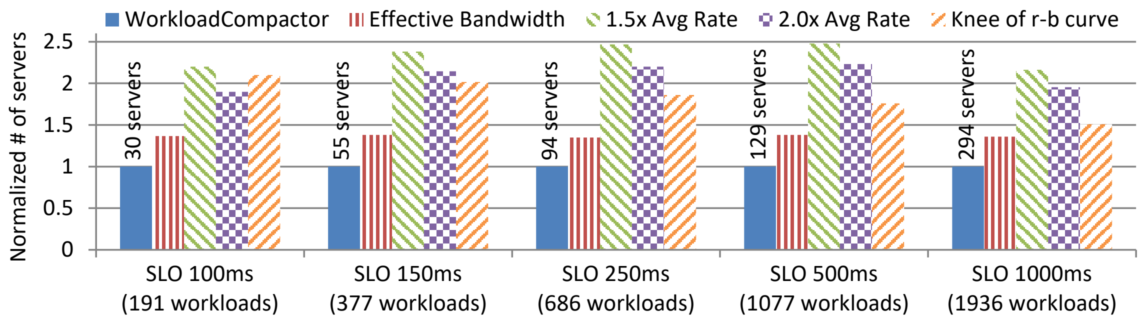


Figure 5.8: Scaling the Same SLO experiments in Figure 5.4 to all available workloads[5]. WorkloadCompactor continues to outperform the state-of-the-art approaches at larger scales.

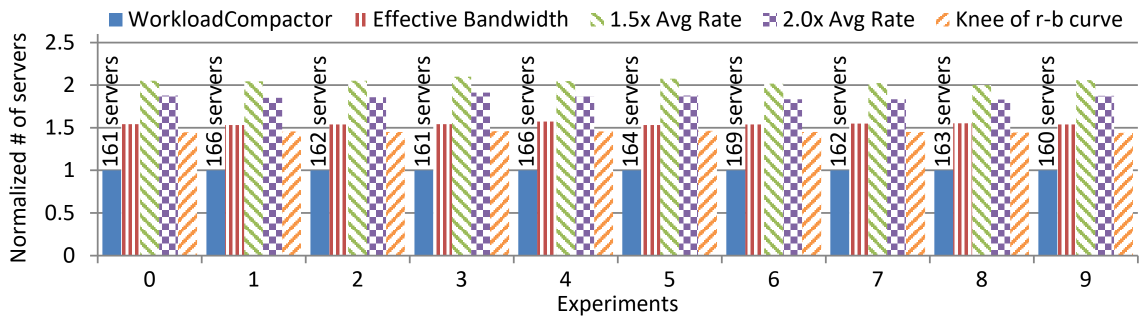


Figure 5.9: Scaling the Random SLO experiment in Figure 5.4 to 1000 workloads. We repeat the experiment with ten random sets of 1000 workloads to show that WorkloadCompactor consistently outperforms state-of-the-art approaches.

scales much better than the typical naïve first-fit policy since FFF skips servers that are nearly full.

One may be concerned about the quality of FFF's packing, since it uses an approximation to check if servers are full. In our experiments, however, we find that FFF produces good packings that only use 3-4% more servers than naïve first-fit while using significantly less computation.

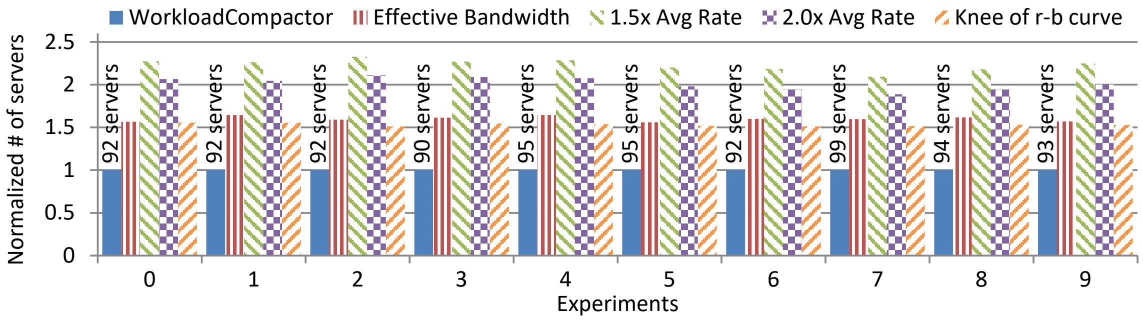


Figure 5.10: Same experiment as Figure 5.9, except with workloads randomly arriving and departing over time. Results measure the maximum number of servers used at any point in time, normalized to WorkloadCompactor. Comparing results to Figure 5.9, we see that WorkloadCompactor handles workload departures better than other approaches since WorkloadCompactor's dynamic reconfiguration naturally adapts to departures.

5.4.4 Scalability of results

Figure 5.8 and Figure 5.9 show the results from scaling the experiments from our local cluster experiments in Section 5.4.1 to more workloads. Our results show that Workload-Compactor's packing density is not significantly affected by the size of the cluster, and we expect WorkloadCompactor to perform well regardless of the cluster size.

5.4.5 Effect of workload departures

So far, we've assumed workloads only arrive over time. In reality, workloads will also depart from the system, leaving gaps in which to place future workloads. To mimic this behavior, we run an experiment where workloads randomly arrive and depart from the system. Our results in Figure 5.10 show that WorkloadCompactor is better able to cope with workload departures than the state-of-the-art approaches, which use over 50% more servers. By contrast, the state-of-the-art approaches use over 40% more servers in the arrival-only scenario in Figure 5.9. This is because WorkloadCompactor can dynamically reconfigure rate limits for previously placed workloads to better pack in new workloads, whereas the other approaches have less flexibility in squeezing in new workloads once a given workload has departed.

[5]Since we only select workloads that can meet its SLO when run in isolation, there are more available workloads with higher SLOs.

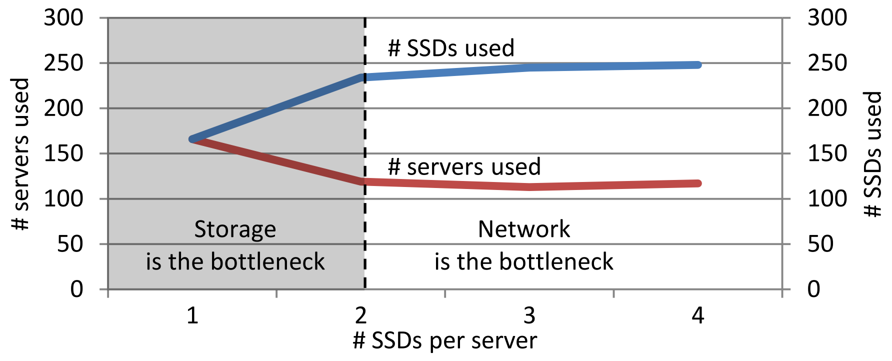


Figure 5.11: The effect of changing the number of SSDs per server in an experiment with 1000 random workloads, each with random SLOs. With 1 SSD per server, the storage is a bottleneck. With 2+ SSDs per server, the network becomes a bottleneck, causing the number of servers and number of SSDs used to plateau. Since WorkloadCompactor accounts for both network and storage, it naturally detects that it doesn't need to use the extra SSDs per server since the network is fully loaded.

5.4.6 Multiple SSDs on a server shift storage bottleneck to network bottleneck

While storage is often a bottleneck, the network can also become a bottleneck depending on the number and bandwidth of SSDs vs. the network bandwidth. Figure 5.11 shows an experiment where we vary the number of SSDs per server to demonstrate this effect. When storage is a bottleneck, increasing the number of SSDs per server should decrease the number of servers used. Eventually, adding more SSDs per server does not help, since now the network has become the bottleneck. For example, in our system, we see that storage is a bottleneck with a single SSD per server, but the network becomes a bottleneck with 2+ SSDs per server. With 2+ SSDs per server, the number of servers used plateaus at around 115 servers, and the number of SSDs used also plateaus since the extra SSDs aren't helpful. In systems with higher network bandwidth, we would expect similar trends, except with the plateau occurring at a higher number of SSDs per server. Importantly, WorkloadCompactor is designed to account for both storage and network, and it will pack workloads so as to not overload either storage or network.

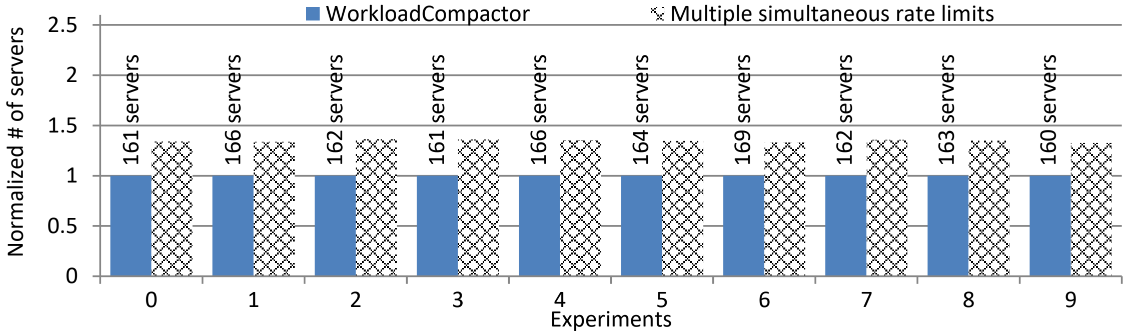


Figure 5.12: Comparing WorkloadCompactor to PriorityMeister's [86] approach of using multiple simultaneous rate limits.

5.4.7 Comparison to using multiple simultaneous rate limits

In addition to the state-of-the-art approaches for selecting rate limits, there is the PriorityMeister [86] approach to use multiple rate limiters simultaneously for each stage (e.g., storage, network) in a workload. Ideally, using multiple simultaneous rate limits will achieve a similar benefit to dynamically reconfiguring rate limits, but there are multiple caveats. First, enforcing multiple simultaneous rate limits is uncommon in systems today, making it harder to deploy. Second, the complexity in analyzing tail latency with multiple rate limits leads to $15\times$ more computation time than WorkloadCompactor with 1000 workloads. Third, the complexity also leads to the analysis being overly conservative when handling equal priority workloads, which results in using more servers than necessary. Consequently, WorkloadCompactor uses fewer servers than the multiple simultaneous rate limits approach, as seen in Figure 5.12.

5.5 Related work

WorkloadCompactor is related to three branches of work, and is the first system to address all three areas. WorkloadCompactor solves the **workload placement** problem in the context of **meeting tail latency SLOs** by optimizing the **selection of rate limit parameters**. Table 5.1 summarizes the differences between related works.

		Workload placement	Tail latency SLOs	Rate limit configuration
Load balancing and migration	Basil [37]	✓	✗	✗
	Pesto [39]	✓	✗	✗
	Romano [61]	✓	✗	✗
	VectorDot [68]	✓	✗	✗
	Delphi/Pythia [23]	✓	✓	✗
Guaranteeing tail latency SLOs	Silo [43]	✓	✓	✗
	QJump [34]	✗	✓	✗
	PriorityMeister [86]	✗	✓	✓
	SNC-Meister [87]	✗	✓	✗
Rate limit configuration	Effective bandwidth [51]	✗	✓	✓
	WorkloadCompactor	✓	✓	✓

Table 5.1: Comparison of WorkloadCompactor's related work.

Workload placement

There are many works that consider how to place and migrate workloads between servers [23, 37, 39, 61, 68]. Many of these works propose good ideas for how to improve latency and throughput with better load balancing [37, 39, 61, 68]. However, ensuring that tail latency SLOs are met is outside the scope of their work.

Delphi/Pythia [23] looks at migrating workloads to meet tail latency SLOs. It reacts to SLO violations and learns the appropriate mitigation actions (e.g., which tenant to migrate). A major limitation is that at the core of its design, it allows SLO violations to occur and then reacts. By contrast, WorkloadCompactor is designed to avoid SLO violations rather than fix bad placements.

Tail latency SLOs

There are four systems that provide tail latency SLO guarantees: Silo [43], QJump [34], PriorityMeister [86], and SNC-Meister [87]. Like WorkloadCompactor, they all use mathematical analysis to ensure SLOs can be met.

Of these works, Silo is the only system that addresses workload placement. The authors find that a first-fit policy works well to pack workloads onto servers. However, Silo does not address how to set rate limits, and the key finding in our work is that the choice of rate limits significantly impacts the ability to compact workloads onto servers. Furthermore, WorkloadCompactor also introduces the fast first-fit feature that drastically improves the computational scalability of workload placement (see Section 5.4.3).

Of these works, PriorityMeister is the only system that considers how to select rate limits. PriorityMeister introduces the idea of simultaneously using multiple rate limiters to avoid picking a specific $\langle r, b \rangle$ rate limit. Conceptually, the idea should work well, but as described in Section 5.4.7, there are multiple caveats that make WorkloadCompactor a superior solution. Additionally, workload placement is outside the scope of the PriorityMeister paper.

Selection of rate limit parameters

Little is known about selecting rate limit parameters since most works (e.g., [34, 35, 43, 85]) assume the user is responsible for selecting rate limits. Users end up relying upon ad hoc heuristics such as scaling the average rate by a factor [43]. The state-of-the-art from theory is an idea known as effective bandwidth [51], described in Section 5.3.1. Though effective bandwidth is optimal when workloads have the same SLO and only traverse a single stage (e.g., storage), our experiments show that WorkloadCompactor uses far fewer servers than effective bandwidth when handling multiple stages or workloads with different SLOs.

5.6 Chapter summary

This chapter considers how to consolidate networked storage workloads onto storage servers while meeting tail latency SLOs. To ensure workloads behave well together, a common approach is to assign rate limits to workloads. Surprisingly, we find that the selection of workload rate limits makes a big difference in the ability to pack workloads together. Unfortunately, there has been little study on how to set rate limits.

WorkloadCompactor introduces a new technique for optimizing the selection of rate limits to compact more workloads onto a server while meeting SLOs. To guarantee tail latency SLOs, WorkloadCompactor enforces rate limits and priorities in storage and network and uses Deterministic Network Calculus (DNC) equations to check if workloads can be placed together while meeting their SLOs. To optimally choose rate limits, we find that WorkloadCompactor needs to adapt the existing workloads' rate limits to better compact them with new workloads that arrive. Our compaction technique is used in conjunction with our scalable placement algorithm, which makes workload placement decisions an order of magnitude faster than the traditional first-fit policy. Experiments with assigning 1000 workloads to servers show that WorkloadCompactor is superior to state-of-the-art approaches, which use 40-150% more servers than WorkloadCompactor.

While we study workload placement in this chapter, we believe our techniques are also useful for the admission control problem. Workload placement is a broader problem than admission control since it additionally involves the decision of where to place workloads.

Hence, our earlier work, SNC-Meister, studies admission control and measures the number of admitted workloads whereas this work studies workload placement and measures the number of servers used.

Beyond workload placement, our techniques naturally extend to other problems such as workload migration. When specifying a new trace in response to a change in an existing workload's behavior, WorkloadCompactor could be used to decide where to migrate the workload if necessary. While we only evaluate WorkloadCompactor with SSDs, WorkloadCompactor can be applied to other storage devices such as disks, and we believe our placement heuristic can be extended to automatically choose resources in heterogeneous storage environments with both SSDs and disks.

WorkloadCompactor is designed for cloud settings where workloads may be adversarial or correlated. By contrast, SNC-Meister in the previous chapter is designed for environments where workloads are generally independent. Both scenarios are important, and our collective works address both scenarios. As for PriorityMeister, WorkloadCompactor draws upon some ideas in the original work and is meant as a replacement. Nevertheless, in scenarios where workloads are already placed and fixed, PriorityMeister is still useful for trying to meet SLOs as best as possible.

Chapter 6

Conclusion

With the recent growth in cloud computing, significant economies of scale are now possible due to widespread sharing of computing and storage resources. Sharing is necessary to amortize the cost of running large datacenters, but it also introduces many performance challenges. In particular, meeting tail latency (e.g., 99th percentile) Service Level Objectives (SLOs) is one of today's hardest and most important problems in resource management. This thesis addresses some of the key challenges in meeting tail latency SLOs when sharing storage and network resources. For example, tail latency is significantly impacted by the transient queues that build up when multiple bursty workloads share resources. In our work, we introduce new techniques for meeting each workload's individual tail latency SLO. We now summarize the contributions made by this thesis and then present some opportunities for future work that arise from our work.

6.1 Contributions

6.1.1 System architecture

In Chapter 2, we describe our system architecture and the key mechanisms for enforcing tail latency SLOs: prioritization and rate limiting. Prioritization allows our system to provide better latency to workloads with tighter SLO requirements, and rate limiting prevents starvation of lower priority workloads. In the chapter, we address the challenges in enforcing priorities and rate limits with different types of hardware (e.g., SSDs, disks). For example, we need to profile the behavior of storage devices to define the notion of tokens in token bucket rate limiting. In the chapter, we also describe the high level process of optimizing the workload priorities and rate limits to meet tail latency SLOs. A key step in this process is analyzing the burstiness of workloads, and we present an algorithm for doing so by analyzing a trace of workload requests. Our system is used in the subsequent chapters to solve resource management questions in meeting tail latency SLOs.

6.1.2 PriorityMeister: Tail latency QoS for shared networked storage

In Chapter 3, we present the design and implementation of a storage and network QoS system, PriorityMeister, for meeting tail latency SLOs. Since existing reactive approaches are unable to cope with the burstiness found in production workloads, PriorityMeister takes a novel approach by proactively analyzing workload behaviors to determine the right QoS parameters for meeting SLOs. Specifically, PriorityMeister automatically configures workload priorities and rate limits based on an analysis of tail latency with Deterministic Network Calculus (DNC). DNC is a powerful mathematical framework for calculating the worst-case latency in a network of queues, and it has been used in two other QoS systems, Silo [43] and QJump [34], since our publication. DNC is the key tool in allowing our algorithms to determine the effect of prioritization and rate limiting on tail latency. Experiments with production workload traces on our physical cluster testbed demonstrate that our approach indeed meets tail latency SLOs, whereas state-of-the-art approaches do not in some cases. Beyond storage and networks, PriorityMeister's techniques can also be extended to analyze latency in real-time systems, where prioritization is common and strict guarantees are desired.

6.1.3 SNC-Meister: Admitting more workloads with tail latency SLOs

In Chapter 4, we build a new admission control system, SNC-Meister, for tail latency SLOs. Admission control is necessary to control the amount of congestion within the system and ensure SLOs are met for admitted workloads. To determine whether tail latency SLOs can be met, SNC-Meister employs a new *probabilistic* theory called Stochastic Network Calculus (SNC), which analyzes tail latency at any percentile (e.g., 99.9%) in a network of queues. SNC-Meister is novel in that it is the first to bring SNC to practice in a computer system. As the first to bring SNC to practice in computer systems, we identify and resolve multiple practical issues such as handling workload dependencies. Our SNC library is now publicly open-sourced at `https://github.com/timmyzhu/SNC-Meister`. Our experiments with production traces show that an SNC approach allows SNC-Meister to admit two to three times more workloads than the more conservative Deterministic Network Calculus (DNC) *adversarial worst-case* approach while still meeting SLOs. We believe the benefits of SNC can extend to other problems beyond admission control such as datacenter provisioning and workload placement in the context of tail latency, and our SNC library provides a solid foundation for future research.

6.1.4 WorkloadCompactor: Reducing datacenter cost while providing tail latency SLO guarantees

In Chapter 5, we introduce a system, WorkloadCompactor, for consolidating workloads onto storage servers while meeting tail latency SLOs. To meet tail latency SLOs, WorkloadCompactor limits the impact of workloads on each other by assigning rate limits and priorities to workloads. Surprisingly, we find that the choice of workload rate limits significantly impacts the ability to pack workloads together. WorkloadCompactor introduces a new technique for optimizing the selection of rate limits to compact more workloads onto a server while meeting SLOs. WorkloadCompactor dynamically reoptimizes rate limits as new workloads arrive to better pack workloads together. Our compaction technique is used in conjunction with our scalable placement algorithm, which makes workload placement decisions an order of magnitude faster than the traditional first-fit policy. Experiments with assigning 1000 workloads to servers show that WorkloadCompactor uses 30-60% fewer servers than state-of-the-art approaches. Beyond workload placement, our techniques can be modified to solve related problems such as workload migration and managing heterogeneous storage environments (e.g., mixture of SSDs and disks).

Though our experiments test 1000-2000 workloads, we expect our techniques to scale to tens of thousands of workloads while deciding workload placements within seconds. If computation time is a limiting factor, the computation can easily be parallelized for each server.

6.2 Future work

This thesis introduces several techniques for meeting tail latency SLOs from the perspective of scheduling policies, admission control, and workload placement. There are two immediate extensions to our work that were not addressed in this thesis.

First, our work on workload placement can immediately be extended to workload migration. This can be accomplished by specifying a new workload trace to our system when an existing workload's behavior changes, at which point our system could decide to migrate the workload if necessary. We have not, however, considered if it is more efficient to migrate other workloads. We also have not explored how to utilize our system to manage the traffic required to migrate a workload's data.

Second, this thesis focuses on SSDs and disks, but does not address other storage devices such as RAID arrays. Our system can be applied to other storage devices with the addition of a profiler (Section 2.3.2) designed for characterizing their performance. Some modifications to our storage enforcement may also be required to handle peculiarities of these storage devices. For example, RAID arrays are similar to SSDs in that they require many concurrent requests to achieve good performance. To handle this peculiarity, we may need to apply techniques similar to the ones used in our SSD enforcement.

Our system introduces new practical ways of utilizing Deterministic Network Calculus (DNC) and Stochastic Network Calculus (SNC) theory. Our theoretically grounded techniques for controlling tail latency can be extended beyond storage and networks to other contexts such as the CPU, cache, etc. For example, our DNC analysis and automatic QoS parameter configuration techniques could apply to real-time CPU scheduling contexts, where prioritization is common and strict guarantees are desired. As another example, our tail latency analysis could be used to build an intelligent storage cache that is aware of the performance impact of its caching decisions on the resulting backend storage traffic.

Our system is designed to ensure upper bounds on tail latency by limiting which workloads can share servers (i.e., admission control/workload placement). This is complementary to other common techniques for reducing tail latency such as replicating requests. We believe request replication is still a useful technique that can be used in our system to reduce tail latency, but our system is still needed to ensure there isn't too much congestion/queueing within the system. In a sense, request replication is designed to find short queues, whereas our system is designed to upper bound the amount of queueing within the system. Extending our system to explicitly account for request replication is left to future work.

Finally, to monetize our work in cloud infrastructures, an important open problem is how to set prices for tail latency SLOs. Designing a pricing model for tail latency is challenging since the price needs to simultaneously account for the SLO latency, the SLO percentile, and the rate limits associated with a workload. In particular, tail latency (and latency in general) is temporal and significantly affected by the burstiness of workloads sharing the system. Our work introduces techniques for characterizing workload burstiness (*r*-*b* curves in Section 2.3.1), but future work is needed to build a pricing model around burstiness and tail latency.

Appendix A

SNC-Meister details and proofs

This appendix gives a detailed explanation of SNC-Meister's analysis technique and the corresponding proof of correctness. In order to state this proof, we first introduce basic SNC definitions and assumptions (Appendix A.1) and the SNC operators (Appendix A.2). We then give a detailed example explaining prior approaches to SNC network analysis and our approach in SNC-Meister (Appendix A.3). Finally, we describe the SNC-Meister analysis algorithm (Appendix A.4) and provide the corresponding correctness proofs (Appendix A.5).

Appendix A.2 and Appendix A.3 describe material that is already known to the SNC community. Appendix A.4 and Appendix A.5 and parts of Appendix A.3 describe material that forms new contributions. These are new techniques that SNC-Meister develops to extend SNC both with respect to making it practical for real systems and also with respect to greatly improving the accuracy of latency bounds derived in SNC.

A.1 Basic SNC assumptions and definitions

Our SNC model is based on the "$\rho(\theta)$, $\sigma(\theta)$" notation developed by Chang [14] and the moment-generating function framework by Fidler [25]. Note that we use the common discrete-time form, where the time step size is small enough to capture continuous-time effects. An excellent in-depth introduction of the discrete-time SNC building blocks and SNC operators can be found in a recent survey [27].

SNC is based on four definitions (the arrival process, the MGF-arrival bound, the service process, and the MGF-service bound), which are modified via the SNC operators (Appendix A.2).

We first formally define the *arrival process*, which captures the total work arriving from a workload in any time interval.

Definition 8. (Arrival process)

Let a_i for $i \geq 0$ denote the work increments of a workload (i.e., the work arriving at time i). The cumulative work received between time m and n is called the arrival process of this workload and is defined:

$$A(m,n) := \sum_{i=0}^{n} a_i - \sum_{i=0}^{m} a_i$$

Using this definition, we can formulate an upper bound on the distribution of the arrival process, using its moment-generating function (MGF). Recall that the MGF of a random variable X is defined as $\mathbb{E}[e^{\theta X}]$.

Definition 9. (MGF-arrival bound)

Let $A(m,n)$ denote the arrival process of a workload. Then, this workload has the MGF-arrival bound $(\rho_A(\theta), \sigma_A(\theta))$, if the moment-generating function of A exists and is upper bounded:

$$\mathbb{E}[e^{\theta A(m,n)}] \leq e^{\theta((n-m)\cdot\rho_A(\theta)+\sigma_A(\theta))} \quad \text{for all } m \leq n \in \mathbb{N} \text{ and } \theta > 0$$

Note that the MGF-arrival bound captures both the burstiness of arrivals and each arrival's request size and thus upper bounds the total work (e.g., bytes) arriving in an interval. A MGF-arrival bound for a Markov-modulated process can be found in Section 4.2.1.

Having bounded a workload's arrivals, we next formalize the service model. We first formally define the *service process* assumption, which formalizes the relation between queue departures and the service process: if there are waiting arrivals, then the minimal number of finished requests (departures) is given by the service process. Note that the service process assumption is also known as the *dynamic server* assumption in the SNC literature [27]. We use $D(m,n)$ to describe the departures (a.k.a. output) from a queue between time m and n (see [26] for more details about this definition).

Definition 10. (Service process (dynamic server))

Let $S(m,n)$ describe the total work processed by a queue between time m and n, and let $D(m,n)$ denote the queue's departures. S is called a service process with departures $D(m,n)$, if S is positive and increasing in n and if for any workload with arrival process $A(m,n)$ it holds that

$$D(0,n) \geq \min_{0 \leq k \leq n} \{A(0,k) + S(k,n)\}$$

Note that the service process assumption is fundamental for the correctness of SNC calculations, and checking this assumption is a key step in the correctness proofs in Appendix A.5. Using the service process definition, we can formulate an upper bound on the distribution of the service process, using its MGF.

Definition 11. (MGF-service bound)

Let S be a service process. Then, S has the MGF-service bound $(\rho_S(\theta), \sigma_S(\theta))$, if the moment-generating function of S exists and is bounded:

$$\mathbb{E}[e^{-\theta S(m,n)}] \leq e^{\theta((n-m)\cdot\rho_S(\theta)+\sigma_S(\theta))} \quad \text{for all } m \leq n \in \mathbb{N} \text{ and } \theta > 0$$

Note that the negative θ in the bound on the MGF actually makes this a lower bound on the service process. As a result, the rate $\rho_S(\theta)$ is also negative.

A.2 Formal definition of the SNC operators

The concepts behind the SNC operators are described in Section 4.2.1. Recall that there are five SNC operators: the latency operator (*Latency*), the leftover operator ($\ominus$), the output operator ($\oslash$), the aggregation operator ($\oplus$), and the convolution operator ($\otimes$). While Table 4.1 gives an overview over the most commonly used form of the operators, this section states the precise mathematical definitions and assumptions and gives pointers to respective correctness proofs in the literature.

In all five of these operators, there is both a dependent version and an independent version. The dependent version works in the case where the arrival/service processes are stochastically dependent (i.e., potentially adversarially correlated), but leads to a worse (i.e., higher) latency prediction than the independent version. Thus, it is preferable to use the independent version whenever processes are independent. A key contribution in our analysis approach is to avoid introducing "artificial" dependencies to minimize the usage of the dependent equations. Appendix A.3 gives an example with details.

We start with the SNC latency operator.

Theorem 1. *(Latency operator [10, 25])*

Let A be an arrival process, and let S be a service process. Assume that A has MGF-arrival bound $(\rho_A(\theta), \sigma_A(\theta))$, S has MGF-service bound $(\rho_S(\theta), \sigma_S(\theta))$, and that $-\rho_S(\theta) > \rho_A(\theta)\ \forall\theta > 0$.

dependent case:

An upper bound on the tail latency as a function of the percentile p is given by:

$$Latency(p) \leq \min_{\theta>0}\Big\{\frac{1}{\theta\rho_S(y\,\theta)}\log\Big((1-p)\cdot\big(1-e^{\theta(\rho_A(x\,\theta)+\rho_S(y\,\theta))}\big)\Big) - \frac{1}{\rho_S(y\,\theta)}(\sigma_A(x\,\theta)+\sigma_S(y\,\theta))\Big\}$$

for any $x,y \in (1,\infty)$ with $\frac{1}{x}+\frac{1}{y}=1$.

independent case:

If A and S are stochastically independent, then the tail latency bound simplifies to:

$$Latency(p) \le \min_{\theta>0}\left\{\frac{1}{\theta\rho_S(\theta)}\log\Big((1-p)\cdot\big(1-e^{\theta(\rho_A(\theta)+\rho_S(\theta))}\big)\Big)-\right.$$
$$\left.\frac{1}{\rho_S(\theta)}(\sigma_A(\theta)+\sigma_S(\theta))\right\}$$

Note that the assumption $-\rho_S(\theta) > \rho_A(\theta)\ \forall\theta > 0$ is essentially a stability condition. ρ_A and ρ_S are time-dependent (i.e., multiplied by $(n-m)$ in Definition 9 and Definition 11) and can be thought of as rates or bandwidths. Also note that the tail latency bound is valid for any fixed $\theta > 0$, and thus the latency operator equation minimizes over all $\theta > 0$. This is done automatically by SNC-Meister as explained in Section 4.2.5.

Finally, note that the dependent case has additional parameters (x and y), besides θ. The latency bound is valid for any x and y (fulfilling $x,y \in (1,\infty)$ with $\frac{1}{x}+\frac{1}{y}=1$), which requires an additional search for the minimal parameters. Additionally, we remark that the dependent case leads to significantly higher latency bounds because there is less multiplexing benefit. Mathematically, the lack of independence means that the dependent-case form relies on the Hölder bound, which is "costly" and leads to a much higher latency prediction [25]. Appendix A.3 explains this further.

The next operator characterizes the leftover service for a workload that shares a queue with a higher-or-equal priority workload.

Theorem 2. *(Leftover operator [10, 25])*

Assume that two workloads share a queue with service process S, for which the first workload has higher or equal priority than the second. Let A_1 and A_2 be the workloads' arrival processes, respectively. Then the service offered by the queue to the second workload A_2 is a service process denoted $S \ominus A_1$.

Assume that A_1 has MGF-arrival bound $(\rho_{A_1}(\theta), \sigma_{A_1}(\theta))$ and that S has MGF-service bound $(\rho_S(\theta), \sigma_S(\theta))$.

dependent case:

The service process $S \ominus A_1$ has the MGF-service bound $(\rho_{S\ominus A_1}(\theta), \sigma_{S\ominus A_1}(\theta))$ given by:

$$\rho_{S\ominus A_1}(\theta) = \rho_{A_1}(x\,\theta) + \rho_S(y\,\theta)$$
$$\sigma_{S\ominus A_1}(\theta) = \sigma_{A_1}(x\,\theta) + \sigma_S(y\,\theta)$$

for any $x,y \in (1,\infty)$ with $\frac{1}{x}+\frac{1}{y}=1$ and for any $\theta > 0$.

independent case:

If A_1 and S are stochastically independent, then the MGF-service bound simplifies to:

$$\rho_{S\ominus A_1}(\theta) = \rho_{A_1}(\theta) + \rho_S(\theta)$$
$$\sigma_{S\ominus A_1}(\theta) = \sigma_{A_1}(\theta) + \sigma_S(\theta)$$

for any $\theta > 0$.

Note that if the queue is shared between many workloads, this theorem can be repeatedly applied because the resulting $S \ominus A_1$ again fulfills the assumption of the theorem.

We also remark, that Theorem 2 is conservative for the case when the two workloads have the same priority. For specific cases of scheduling policies, like FIFO scheduling, there are more accurate analysis techniques in the literature [18]. However, since switching fabrics do not strictly follow FIFO in practice, our analysis does not rely on assuming a specific scheduling policy (such as FIFO).

The next operator is the output operator, which is used to calculate a bound on the departures from a queue, which can then form the input (arrival process) to another queue in a network.

Theorem 3. *(Output operator [10, 25])*

Suppose a workload with arrival process A traverses a queue with service process S. Then the ("output") departure process is an arrival process denoted $A \oslash S$.

Assume that A has MGF-arrival bound $(\rho_A(\theta), \sigma_A(\theta))$ and that S has MGF-service bound $(\rho_S(\theta), \sigma_S(\theta))$.

dependent case:

The departure process $A \oslash S$ has the MGF-arrival bound $(\rho_{A\oslash S}(\theta), \sigma_{A\oslash S}(\theta))$ given by:

$$\rho_{A\oslash S}(\theta) = \rho_A(x\,\theta)$$
$$\sigma_{A\oslash S}(\theta) = \sigma_A(x\,\theta) + \sigma_S(y\,\theta) - \frac{1}{\theta}\log\left(1 - e^{\theta(\rho_A(x\,\theta)+\rho_S(y\,\theta))}\right)$$

for any $x, y \in (1, \infty)$ with $\frac{1}{x} + \frac{1}{y} = 1$ and for any $\theta > 0$.

independent case:

If A and S are stochastically independent, then the MGF-arrival bound simplifies to:

$$\rho_{A\oslash S}(\theta) = \rho_A(\theta)$$
$$\sigma_{A\oslash S}(\theta) = \sigma_A(\theta) + \sigma_S(\theta) - \frac{1}{\theta}\log\left(1 - e^{\theta(\rho_A(\theta)+\rho_S(\theta))}\right)$$

for any $\theta > 0$.

The next SNC operator is the aggregation operator, which is used to merge two arrival processes into one.

Theorem 4. *(Aggregation operator)*

Let A_1 and A_2 be two arrival processes. Then the multiplexed arrival process is an arrival process denoted $A_1 \oplus A_2$.

Assume that A_1 has MGF-arrival bound $(\rho_{A_1}(\theta), \sigma_{A_1}(\theta))$ and that A_2 has MGF-arrival bound $(\rho_{A_2}(\theta), \sigma_{A_2}(\theta))$.

dependent case:

The aggregated arrival process $A_1 \oplus A_2$ has the MGF-arrival bound $(\rho_{A_1 \oplus A_2}, \sigma_{A_1 \oplus A_2})$ given by:

$$\rho_{A_1 \oplus A_2}(\theta) = \rho_{A_1}(x\,\theta) + \rho_{A_2}(y\,\theta)$$
$$\sigma_{A_1 \oplus A_2}(\theta) = \sigma_{A_1}(x\,\theta) + \sigma_{A_2}(y\,\theta)$$

for any $x, y \in (1, \infty)$ with $\frac{1}{x} + \frac{1}{y} = 1$ and for any $\theta > 0$.

independent case:

If A_1 and A_2 are stochastically independent, then the MGF-arrival bound simplifies to:

$$\rho_{A_1 \oplus A_2}(\theta) = \rho_{A_1}(\theta) + \rho_{A_2}(\theta)$$
$$\sigma_{A_1 \oplus A_2}(\theta) = \sigma_{A_1}(\theta) + \sigma_{A_2}(\theta)$$

for any $\theta > 0$.

The final operator is the convolution operator, which is used to "merge" two (or more) queues in sequence into a single mathematical representation.

Theorem 5. *(Convolution operator [25])*

Let S_1 and S_2 be service processes for two queues in sequence. Then the combined effect of both queues is a service process denoted $S_1 \otimes S_2$.

Assume that S_1 has MGF-service bound $(\rho_{S_1}(\theta), \sigma_{S_1}(\theta))$, S_2 has MGF-service bound $(\rho_{S_2}(\theta), \sigma_{S_2}(\theta))$, and that $\rho_{S_1}(\theta) \neq \rho_{S_2}(\theta)$.

dependent case:

The convoluted service process $S_1 \otimes S_2$ has the MGF-service bound $(\rho_{S_1 \otimes S_2}, \sigma_{S_1 \otimes S_2})$ given by:

$$\rho_{S_1 \otimes S_2}(\theta) = \max\left\{\rho_{S_1}(x\,\theta), \rho_{S_2}(y\,\theta)\right\}$$
$$\sigma_{S_1 \otimes S_2}(\theta) = \sigma_{S_1}(x\,\theta) + \sigma_{S_2}(y\,\theta) - \frac{1}{\theta}\log\left(1 - e^{-\theta\left|\rho_{S_1}(x\,\theta) - \rho_{S_2}(y\,\theta)\right|}\right)$$

for any $x, y \in (1, \infty)$ with $\frac{1}{x} + \frac{1}{y} = 1$ and for any $\theta > 0$.

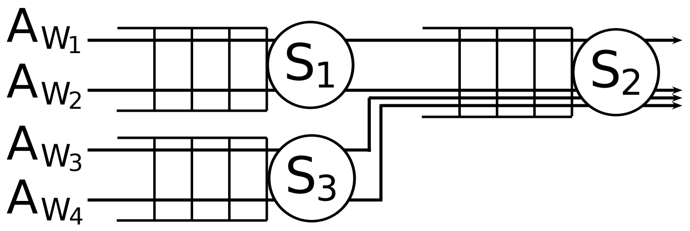


Figure A.1: Example network with four workloads W_1 to W_4 flowing through three queues S_1, S_2, and S_3.

independent case:
If S_1 and S_2 are stochastically independent, then the MGF-service bound simplifies to:

$$\rho_{S_1 \otimes S_2}(\theta) = \max\left\{\rho_{S_1}(\theta), \rho_{S_2}(\theta)\right\}$$
$$\sigma_{S_1 \otimes S_2}(\theta) = \sigma_{S_1}(\theta) + \sigma_{S_2}(\theta) - \frac{1}{\theta}\log\left(1 - e^{-\theta\left|\rho_{S_1}(\theta) - \rho_{S_2}(\theta)\right|}\right)$$

for any $\theta > 0$.

The idea behind the convolution theorem is that it can be repeatedly applied until each workload's arrival process in a network traverses a single (convolution-type) service process [17, 25].

Note that the case where $\rho_{S_1}(\theta) = \rho_{S_2}(\theta)$ is not covered by this theorem. The simplest way around this problem is to assume that one of the servers is slightly slower than the other (e.g., scaling $\rho_{S_1}(\theta)$ by 0.99), which makes little difference numerically and allows us to always use this theorem.

A.3 Example: SNC convolution, hop-by-hop, and SNC-Meister analysis

Having formally introduced the SNC operators, we are now ready to give an example for the hop-by-hop analysis technique, convolution analysis technique, and SNC-Meister's analysis technique. All three techniques are based on the five SNC operators, but differ in the order in which the operators are applied.

Recall the example network analyzed in Section 4.2.2, repeated here as Figure A.1. There are four workloads, with arrival processes A_{W_1}, A_{W_2}, A_{W_3}, and A_{W_4}. The four workloads traverse three queues, with service processes S_1, S_2, and S_3. For the sake of simplicity, we assume that W_1 has a strictly lower priority than W_2 to W_4 on all queues and that all workloads are stochastically independent to start with. Appendix A.4 shows how to work with user-specified workload dependencies. We furthermore require that all workloads have MGF-arrival bounds and all service processes have MGF-service bounds.

Suppose we would like to calculate an upper bound on workload W_1's 99th percentile latency. To perform this analysis, we use the SNC operators from Appendix A.2: the latency operator (*Latency*), the leftover operator ($\ominus$), the output operator ($\oslash$), the aggregation operator ($\oplus$), and the convolution operator ($\otimes$).

All network analysis approaches have to first consider the departures from workloads W_3 and W_4 at S_3. A straightforward application of the leftover and output operators at S_3, would calculate their departures from S_3 as follows:

$$A'_{W_3} = A_{W_3} \oslash (S_3 \ominus A_{W_4})$$
$$A'_{W_4} = A_{W_4} \oslash (S_3 \ominus A_{W_3})$$

Then, when analyzing S_2 and subtracting A'_{W_3} and A'_{W_4} from S_2 (to calculate the service available to W_1), we would run into an artificial stochastic dependency because A'_{W_3} and A'_{W_4} are not independent. In this example, it is easy to avoid this artificial dependency by aggregating A_{W_3} and A_{W_4} right from the start. This aggregation trick is a key part of SNC-Meister's analysis technique and will be discussed in more detail later.

We next state the explicit operator sequences to analyze the whole network based on the hop-by-hop approach, convolution approach, and SNC-Meister's approach.

The first approach is called hop-by-hop, because it separately applies the tail latency bound from Theorem 1 to each hop (i.e., queue). Recent work [10] has shown that this technique can be used to analyze a broad set of queueing networks (feed-forward networks).

Analysis approach 1. *(SNC hop-by-hop)*

We first derive the service S'_1 offered to W_1 at the first queue:

$$S'_1 = S_1 \ominus A_{W_2}$$

where we subtract the arrival processes of the workload W_2. We can then calculate the tail latency W_1 at the first queue with:

$$Latency(A_{W_1}, S'_1, 0.995) \tag{A.1}$$

In order to analyze the latency at the second queue, we first derive the arrival process of W_1 at the second queue (which is the departure process from the first queue)

$$A'_{W_1} = A_{W_1} \oslash S'_1$$

using the output operator. Similarly, we derive the departure process for W_2 as $A'_{W_2} = A_{W_2} \oslash S_1$. For workloads W_3 and W_4 we first aggregate them into a single arrival process *and then calculate their departure process from S_3 as $A'_{W_{3/4}} = (A_{W_3} \oplus A_{W_4}) \oslash S_3$. The local service S'_2 offered to W_1 at the second queue is then derived as:*

$$S'_2 = S_2 \ominus A'_{W_2} \ominus A'_{W_{3/4}}$$

and we calculate the tail latency of W_1 at the second queue with:

$$Latency(A'_{W_1}, S'_2, 0.995) \tag{A.2}$$

We add the two latencies in Equation (A.1) *and Equation* (A.2) *to calculate W_1's 99th percentile latency. This is valid by the union bound because each hop's latency uses a higher percentile (e.g., 99.5% here).*

Note that Equation (A.2) *includes a stochastic dependency (e.g., S_1 occurs in both A'_{W_1} and S'_2), which means that the operators cannot use the simplified independent-case equation, but need to use the more complex dependent-case equation.*

Observe that the stochastic dependency in the hop-by-hop analysis approach is inherent to the analysis and not due to actual dependencies between workloads (we assumed them to be initially stochastically independent). We therefore call such a stochastic dependency an *artificial dependency* as opposed to an actual (user-specified) dependency.

The second analysis approach is called SNC convolution and emerges when applying the line-network analysis technique [10, 12, 16, 17, 25, 27, 30, 54] to our network. The goal of the SNC convolution approach is to merge all queues into a single service process and to then apply Theorem 1 once to obtain the tail latency.

Analysis approach 2. *(SNC convolution)*

We first apply the leftover operator to every queue to obtain the "local service process" offered to W_1, denoted S'_1 at the first queue and S'_2 at the second queue. They are calculated exactly the same as in Approach 1.

We next use the convolution operator to merge the two local service processes together into a global service process S:

$$S = \left(S'_1 \otimes S'_2\right) \tag{A.3}$$

We can then calculate the 99th percentile tail latency of W_1 at both queues using:

$$Latency(A_{W_1}, S, 0.99)$$

Note that Equation (A.3) *includes an artificial stochastic dependency. The benefit of the convolution approach is that it only requires a single latency calculation.*

In contrast to both SNC hop-by-hop and SNC convolution, SNC-Meister uses a novel operator sequence, which minimizes the number of stochastic dependencies in the network analysis. For the particular example given here, it is easy to show that our analysis does not have any stochastic dependencies, which leads to a more accurate analysis.

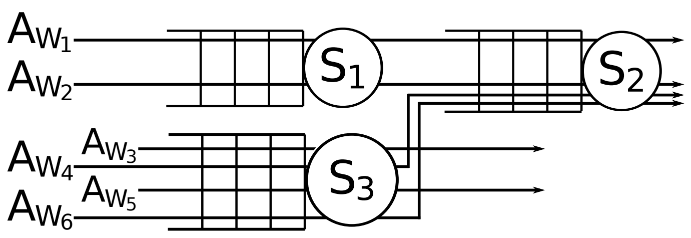


Figure A.2: Reordering of workload priorities can be necessary to allow for aggregation of workloads. In this figure, the workload priorities are ordered $W_3 > W_4 > W_5 > W_6$, which makes aggregating W_4 and W_6 impossible. In order to aggregate W_4 and W_6, they need to be in the same priority class, which means decreasing the priority of W_4 (for the sake of the analysis).

Analysis approach 3. *(SNC-Meister)*

The idea is similar to Approach 2, but changes the position of A_{W_2} in the operator sequence. Specifically, we exclude W_2 in the derivation of the local service processes for each queue, because it shares the whole path with W_1. We apply the leftover operator to W_2 and W_1 only after having merged the two local service processes into a global service process. This leads to the following operator sequence:

$$S = \Big(S_1 \otimes \big(S_2 \ominus ((A_{W_3} \oplus A_{W_4}) \oslash S_3)\big)\Big) \ominus A_{W_2} \tag{A.4}$$

where in the inner-most parenthesis we aggregate W_3 and W_4 before calculating their departures from S_3. We then subtract them from S_2. In the outer-most parenthesis, W_2 is subtracted after having merged S_1 and the leftover from S_2 using the convolution operator. We calculate the tail latency of W_1 at both queues using:

$$Latency(A_{W_1}, S, 0.99)$$

Note that Equation (A.4) *does not include any stochastic dependencies.*

Note that SNC-Meister's equation is simpler than the other approaches, but requires changing the order of several operators, which are not necessarily exchangeable. We therefore prove the correctness of this change in the operator sequence in Appendix A.5.

We remark that the aggregation step (described before the three approaches) play an important role in preventing stochastic dependencies. Unfortunately, as more and more workloads are added to a network, aggregating departures becomes more complex.

Figure A.2 expands the network from Figure A.1 with two additional workloads traversing S_3. Specifically, we now consider four workloads at S_3, W_3 to W_6, of which only two, W_4 and W_6, traverse the queue S_2. The four workloads are ordered by strictly decreasing scheduling priority. As in the previous example, we are interested in analyzing S_2, which requires the departures from W_4 and W_6.

If we calculate the departures of W_4 and W_6 separately (i.e., without aggregation), we would introduce artificial stochastic dependencies. Therefore, it would be helpful to aggregate the departures. Unfortunately, W_4 and W_6 have different scheduling priorities, which prevents aggregating them into a single arrival process. SNC-Meister solves this problem by relaxing the priority of the higher-priority workload, W_4, and then calculating the aggregated arrival process. While this seems at first counter-intuitive – as assuming W_4 has a lower priority makes the analysis conservative – this step is necessary to resolve artificial stochastic dependencies which would be introduced without aggregation. Changing these priorities (for the sake of analysis) and efficiently aggregating workloads are key parts of SNC-Meister's analysis algorithm.

A.4 SNC-Meister's analysis algorithm

This section introduces the two parts of SNC-Meister's analysis algorithm: the arrival process aggregation algorithm and the network analysis algorithm based on the aggregation algorithm.

The first part, the arrival process aggregation algorithm, has the goal of aggregating many arrival processes, which might have inter-dependencies originating from user-specified dependencies. Specifically, the input to this algorithm is a list of arrival processes, and a graph of their dependencies. The dependency graph has an edge between arrival process A_i and A_j, if they are inter-dependent. In SNC-Meister we assume that user-specified dependencies are transitive (i.e., if *i* and *j* are dependent, and *j* and *k* are dependent, then also workloads *i* and *k* must be dependent). This means that the dependency graph consists of several cliques, which each represent one set of inter-dependent arrival processes.

The goal of the aggregation algorithm is to apply the minimal number of dependent-case SNC operators to merge a set of arrival processes into a single arrival process. The algorithm proceeds in four steps:

1. create a list of groups G;
2. for each arrival process A, add A to the lowest numbered group in G that does not have a workload with a dependency on A;
3. for each group g in G, aggregate the arrival processes in g, which are all independent by construction, and store the aggregate in G';
4. aggregate all the aggregates in G', which all are dependent by construction.

The output is a single arrival process.

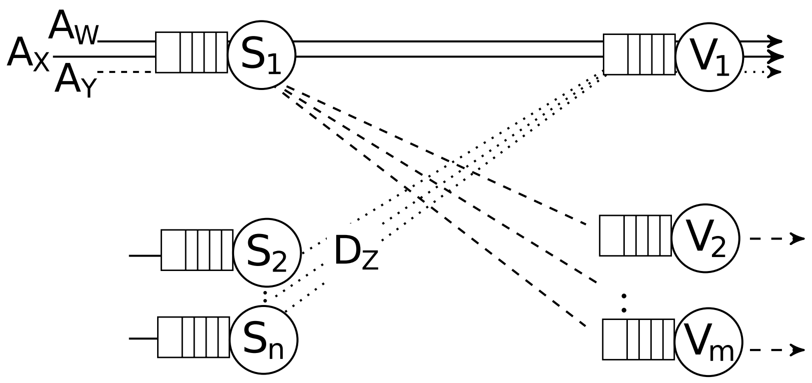


Figure A.3: SNC-Meister calculates upper bounds for a bipartite graph of service processes $S_i, i = 1, ..., n$ (left-hand side) and $V_j, j = 1, ..., m$ (right-hand side). This corresponds to a full bisection bandwidth network where congestion primarily occurs at the end-hosts.

The second part, SNC-Meister's network analysis algorithm, has the goal of deriving an upper bound on the tail latency for a given workload. The algorithm is designed for full bisection bandwidth networks where congestion primarily occurs at the end-hosts. This is represented by a network of queues in a bipartite graph (Figure A.3) with two sets of service processes S_i ($i \in 1, ..., n$) and V_j ($j \in 1, ..., m$). Without loss of generality, we consider the workload labeled W, which traverses S_1 and V_1. SNC-Meister's network analysis algorithm then proceeds in six steps:

1. Find the set X of all workloads (excluding W) that traverse S_1 and V_1 and have a higher or equal priority than W. Use the aggregation algorithm to merge the arrival processes of all workloads in X and denote the aggregated arrival process by A_X.
2. Find the set Y of all workloads that traverse S_1 and V_j ($j > 1$) and have a higher or equal priority than W. Use the aggregation algorithm to merge the arrival processes of all workloads in Y and denote the aggregated arrival process by A_Y.
3. For each $i > 1$:
   - Find all workloads Z_i, which traverse S_i and V_1 and have a higher or equal priority than W. Use the aggregation algorithm to merge all of Z_i's arrival processes into A_{Z_i}.
   - Find all workloads R_i, which traverse S_i, are not in Z_i, and have a higher or equal priority than the lowest priority workload in Z_i. Use the aggregation algorithm to merge all of R_i's arrival processes into A_{R_i}.
   - Calculate the departure process for S_i: $D_{Z_i} = A_{Z_i} \oslash (S_i \ominus A_{R_i})$.

4. Use the aggregation algorithm to merge all departure processes into $D_Z = D_{Z_2} \oplus D_{Z_3} \oplus D_{Z_4} \oplus \ldots \oplus D_{Z_n}$.
5. Calculate the network service process for workload W using the following equation:

$$S = \Big(\big((S_1 \ominus A_Y) \otimes (V_1 \ominus D_Z) \big) \ominus A_X \Big) \tag{A.5}$$

6. Use W's arrival process, S, and the SNC latency bound to derive the latency for W, for the percentile p via $Latency(A_W, S, p)$.

A.5 Correctness of SNC-Meister's analysis algorithm

This section describes the correctness proofs of SNC-Meister's analysis algorithm described in the previous section. Specifically, we prove three statements:

(A) that the aggregation algorithm does in fact lead to the minimal number of stochastic dependencies

(B) that step 3 in the network analysis correctly calculates an output bound

(C) the correctness of the service process equation Equation (A.5)

To prove statement (A), recall the dependency graph, which is the input to the aggregation algorithm. As described in the previous section, this graph consists of several cliques, where each clique represents a set of inter-dependent arrival processes.

Theorem 6. *(Optimality of aggregation algorithm)*

Let k be the maximum size of a clique in the dependency graph.

1. *The minimal number of applications of dependent-case SNC operators for any algorithm is at least $k-1$.*
2. *Our aggregation algorithm requires $k-1$ applications of dependent-case SNC operators.*

Proof. Note that the first statement is trivial because clearly each arrival process in the largest clique has to be aggregated with a dependency operation. Since all workloads in the largest clique (of size k) are inter-dependent, we need at least $k-1$ aggregations with the dependent-case SNC operator.

We next prove that our aggregation algorithm needs at most $k-1$ dependent-case operations. Assume for the sake of contradiction that the aggregation algorithm requires k dependency operations. Our aggregation algorithm only uses dependency operations in step 4, which requires $|G'|-1$ dependency operations. Thus, $k = |G'|-1 = |G|-1$. This implies $|G| = k+1$, which means that in step 2, there was some arrival process A^*, such that A^* was added to the $k+1$ group. This can only happen if A^* is dependent with some arrival process in all groups $1,\ldots,k$. Now by the assumption of transitivity of workload dependencies, we have a clique of size $k+1$ with A^* and the other arrival processes that it is dependent on in each group. This is a contradiction to k being the maximum size of a clique. $\square$

To prove statement (B), we show that decreasing the priority of a workload (for the sake of analysis) leads to an upper bound on the workload's departures.

Lemma 1. *(Aggregation with changed priorities)*

Assume that a set of workloads traverses a queue with service process S. Let Z denote a subset for which we are interested in a bound on the aggregated departures. Let R denote all workloads with an equal or higher priority than the lowest-priority workload in Z. Assume that the aggregated arrival processes from Z and R have MGF-bounds A_Z and A_R, respectively.

Then, the departure process of all Z workloads, D_Z, is upper bounded

$$D_Z \leq A_Z \oslash (S \ominus A_R)$$

Proof. It is sufficient to show that for every workload, calculating the departure process by decreasing the workload's priority is an upper bound on the departure process with the original priority. To this end, let A denote any fixed workload and let S denote the local service process of A at the queue. Let D denote A's departures. According to the departure theorem [51], it holds that

$$D(m,n) \leq \max_{0\leq k\leq m} \{A(k,n) - S(k,m)\} \tag{A.6}$$

Decreasing the priority of A results in a service process $S'(m,n) \leq S(m,n)$ (for all $m \leq n \in \mathbb{N}$). Therefore, $\max_{0\leq k\leq m}\{A(k,n) - S'(k,m)\}$ gives an upper bound on the right-hand side of Equation (A.6). $\square$

To prove statement (C) (i.e., Equation (A.5)), we need to verify the assumptions of the SNC latency bound, which is that S is a service process (Definition 10).

The following theorem formally describes this scenario and SNC-Meister's operator sequence together with a full proof of correctness.

Theorem 7. *(Independent network analysis)*

Assume the scenario shown in Figure A.3:

- *a bipartite graph connecting two sets of service processes S_i ($i \in 1,\ldots,n$) and V_j ($j \in 1,\ldots,m$)*
- *a set of workloads X traverses S_1 and V_1, and the aggregate arrivals are bounded by A_X;*

- *a set of workloads Y traverses S_1 and V_j ($j > 1$), and the aggregate arrivals are bounded by A_Y;*
- *a set of workloads Z originates as departures from S_i ($i > 1$) and traverses V_1, and the aggregate departures from these workloads are bounded by D_Z*

We are interested in workload W, which traverses S_1 and V_1 and is bounded by arrival process A_W. W has a lower or equal priority to X, Y, and Z. The order of priorities between X, Y, Z can be arbitrary.

Then, the tail latency can be calculated using Theorem 1 with A_W and the service process S

$$S(m,n) = \Big(\big((S_1 \ominus A_Y) \otimes (V_1 \ominus D_Z) \big) \ominus A_X \Big)(m,n)$$

whose MGF-service bound is calculated using the standard SNC operators.

Proof. Note that because the correctness of individual operators has already been proven, it remains to be shown that changing the operator sequence satisfies the service process requirement from Definition 10.

Let $n \geq 0$ be arbitrary. Consider W and X. Let D_W and D_X denote their departures from server S_1, and let D_W^* and D_X^* denote their departures from server V_1.

We first consider the relation between the departures of W and X at V_1 (i.e., D_W^*, D_X^*) to their arrivals (i.e., D_W, D_X). By Lemma 1, they receive the service process $V_1 \ominus D_Z$. That is (by Definition 10),

$$D_W^*(0,n) + D_X^*(0,n) \geq \min_{0 \leq m \leq n} \big\{ D_W(0,m) + D_X(0,m) + (V_1 \ominus D_Z)(m,n) \big\} \tag{A.7}$$

Similarly, consider the relation between the departures of W and X at S_1 (i.e., D_W, D_X) to their arrivals (i.e., A_W, A_X). By Lemma 1, they receive the service process $S_1 \ominus A_Y$. That is (by Definition 10),

$$D_W(0,m) + D_X(0,m) \geq \min_{0 \leq k \leq m} \big\{ A_W(0,k) + A_X(0,k) + (S_1 \ominus A_Y)(k,m) \big\} \tag{A.8}$$

Now combining Equation (A.7) and Equation (A.8), we get

$$\begin{aligned}
&D_W^*(0,n) + D_X^*(0,n) \\
&\quad \geq \min_{0 \leq m \leq n} \Big\{ \min_{0 \leq k \leq m} \big\{ A_W(0,k) + A_X(0,k) + (S_1 \ominus A_Y)(k,m) \big\} + (V_1 \ominus D_Z)(m,n) \Big\} \\
&\quad = \min_{0 \leq k \leq m \leq n} \Big\{ A_W(0,k) + A_X(0,k) + (S_1 \ominus A_Y)(k,m) + (V_1 \ominus D_Z)(m,n) \Big\} \\
&\quad = \min_{0 \leq k \leq n} \Big\{ A_W(0,k) + A_X(0,k) + \min_{k \leq m \leq n} \big\{ (S_1 \ominus A_Y)(k,m) + (V_1 \ominus D_Z)(m,n) \big\} \Big\} \\
&\quad = \min_{0 \leq k \leq n} \Big\{ A_W(0,k) + A_X(0,k) + \big((S_1 \ominus A_Y) \otimes (V_1 \ominus D_Z) \big)(k,n) \Big\}
\end{aligned}$$

Next, we note that

$$A_X(0,n) \geq D_X(0,n) \geq D_X^*(0,n)$$

since there must be an arrival for there to be a departure.

Combining the previous inequalities, we get

$$\begin{aligned}
D_W^*(0,n) &\geq \min_{0\leq k\leq n} \Big\{ A_W(0,k) + A_X(0,k) + \big((S_1 \ominus A_Y) \otimes (V_1 \ominus D_Z)\big)(k,n) \Big\} - A_X(0,n) \\
&= \min_{0\leq k\leq n} \Big\{ A_W(0,k) - A_X(k,n) + \big((S_1 \ominus A_Y) \otimes (V_1 \ominus D_Z)\big)(k,n) \Big\} \\
&= \min_{0\leq k\leq n} \Big\{ A_W(0,k) + \Big(\big((S_1 \ominus A_Y) \otimes (V_1 \ominus D_Z)\big) \ominus A_X\Big)(k,n) \Big\} \\
&= \min_{0\leq k\leq n} \Big\{ A_W(0,k) + S(k,n) \Big\}
\end{aligned}$$

Thus, by Definition 10, S is the service process for W. □

Observe that $S(m,n)$ preserves all stochastic independencies of W, X, Y, and Z. SNC-Meister's network analysis is thus optimal in the sense that it does not introduce artificial stochastic dependencies.

Bibliography

[1] Alan Agresti. Building and applying logistic regression models. *Categorical Data Analysis, Second Edition*, pages 211–266, 2007. 4.3.2

[2] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp DCTCP. In *ACM SIGCOMM*, pages 63–74, 2011. 4.5, 4.5

[3] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 19–19, Berkeley, CA, USA, 2012. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2228298.2228324`. 3.5

[4] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *USENIX NSDI*, pages 19–19, 2012. 4.1, 4.5

[5] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM*, pages 435–446, 2013. 4.1, 4.5, 4.5, 5.1

[6] Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst.*, 19(4):483–518, November 2001. ISSN 0734-2071. doi: 10.1145/502912.502915. URL `http://doi.acm.org/10.1145/502912.502915`. 1.7, 1.8

[7] Eric Anderson. Simple table-based modeling of storage devices, 2001. 2.3.2

[8] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running circles around storage administration. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*,

FAST'02, pages 13–13, Berkeley, CA, USA, 2002. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1973333.1973346`. 1.7, 1.8

[9] Yossi Azar, Ilan Reuven Cohen, Seny Kamara, and Bruce Shepherd. Tight bounds for online vector bin packing. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC '13, pages 961–970, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2029-0. doi: 10.1145/2488608.2488730. URL `http://doi.acm.org/10.1145/2488608.2488730`. 3

[10] Michael A Beck and Jens Schmitt. The disco stochastic network calculator version 1.0: when waiting comes to an end. In *Valuetools*, pages 282–285, 2013. 4.7, 4.2.2, 4.5, 1, 2, 3, A.3, A.3

[11] Anne Bouillard, Laurent Jouhet, and Éric Thierry. Tight performance bounds in the worst-case analysis of feed-forward networks. In *IEEE INFOCOM*, pages 1316–1324, 2010. ISBN 978-1-4244-5836-3. 3.2.3, 3.5

[12] Almut Burchard, Jörg Liebeherr, and Florin Ciucu. On superlinear scaling of network delays. *IEEE/ACM Transactions on Networking (TON)*, 19(4):1043–1056, 2011. 4.5, A.3

[13] Cheng-Shang Chang. Stability, queue length, and delay of deterministic and stochastic queueing networks. *IEEE Transactions on Automatic Control*, 39(5):913–931, 1994. 1.7, 4.5

[14] Cheng-Shang Chang. *Performance guarantees in communication networks*. Springer Science & Business Media, 2000. 1.7, 4.5, A.1

[15] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *IEEE HPCA*, pages 266–277, 2011. 1

[16] Florin Ciucu and Jens Schmitt. Perspectives on network calculus: No free lunch, but still good value. In *ACM SIGCOMM*, pages 311–322, 2012. 4.5, A.3

[17] Florin Ciucu, Almut Burchard, and Jörg Liebeherr. A network service curve approach for the stochastic analysis of networks. In *ACM SIGMETRICS*, pages 279–290, 2005. 1.7, 4.5, A.2, A.3

[18] Rene L Cruz. Sced+: Efficient management of quality of service guarantees. In *IEEE INFOCOM*, volume 2, pages 625–634, 1998. A.2

[19] RL Cruz. Quality of service management in integrated services networks. In *Proceedings of the 1st Semi-Annual Research Review, CWC*, 1996. 1.7, 4.5

[20] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2): 74–80, February 2013. ISSN 0001-0782. doi: 10.1145/2408776.2408794. URL

http://doi.acm.org/10.1145/2408776.2408794. 1.1, 4.1, 4.5, 4.5, 5.1

[21] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SOSP*, pages 205–220, 2007. ISBN 978-1-59593-591-5. 1.1, 1.7, 4.1, 4.1, 5.1

[22] Cagdas Dirik and Bruce Jacob. The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 279–289, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0. doi: 10.1145/1555754.1555790. URL http://doi.acm.org/10.1145/1555754.1555790. 1

[23] Aaron J. Elmore, Sudipto Das, Alexander Pucher, Divyakant Agrawal, Amr El Abbadi, and Xifeng Yan. Characterizing tenant behavior for placement and crisis mitigation in multitenant dbmss. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 517–528, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. doi: 10.1145/2463676.2465308. URL http://doi.acm.org/10.1145/2463676.2465308. 1.7, 1.8, 5.4.7, 5.5

[24] Domenico Ferrari and Dinesh C Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE JSAC*, 8(3):368–379, 1990. 4.5

[25] Markus Fidler. An end-to-end probabilistic network calculus with moment generating functions. In *IEEE International Workshop on Quality of Service (IWQoS)*, pages 261–270, 2006. 1.7, 4.7, 4.2.2, 4.5, A.1, 1, A.2, 2, 3, 5, A.2, A.3

[26] Markus Fidler. Survey of deterministic and stochastic service curve models in the network calculus. *IEEE Communications Surveys & Tutorials*, 12(1):59–86, 2010. A.1

[27] Markus Fidler and Amr Rizk. A guide to the stochastic network calculus. *IEEE Communications Surveys & Tutorials*, 17(1):92–105, 2015. 1.8, 4.5, A.1, A.1, A.3

[28] Victor Firoiu, Jean-Yves Le Boudec, Don Towsley, and Zhi-Li Zhang. Theories and models for internet quality of service. *Proceedings of the IEEE*, 90(9):1565–1591, 2002. 1.7, 4.5

[29] 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 *ACM CoNEXT*, 2015. 4.5, 4.5

[30] Yashar Ghiassi-Farrokhfal and Jörg Liebeherr. Output characterization of constant bit rate traffic in fifo networks. *IEEE Communications Letters*, 13(8):618–620, 2009. 4.5,

A.3

[31] Yashar Ghiassi-Farrokhfal, Srinivasan Keshav, and Catherine Rosenberg. Toward a realistic performance analysis of storage systems in smart grids. *IEEE Transactions on Smart Grid*, 6(1):402–410, 2015. 4.5

[32] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 24–24, Berkeley, CA, USA, 2011. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1972457.1972490`. 3.3.1, 3.5

[33] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 1–12, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1419-0. doi: 10.1145/2342356.2342358. URL `http://doi.acm.org/10.1145/2342356.2342358`. 3.5

[34] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can jump them! In *USENIX NSDI*, 2015. 1.7, 3.6, 4, 4.1, 4.1, 4.1, 4.3.1, 4.5, 4.5, 4.5, 5.1, 5.2.1, 5.4.7, 5.5, 5.5, 6.1.2

[35] Ajay Gulati, Arif Merchant, and Peter J. Varman. pclock: an arrival curve based approach for qos guarantees in shared storage systems. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '07, pages 13–24, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-639-4. doi: 10.1145/1254882.1254885. URL `http://doi.acm.org/10.1145/1254882.1254885`. 1.7, 3.1, 3.4.5, 3.5, 5.1, 5.5

[36] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. Parda: proportional allocation of resources for distributed storage access. In *Proccedings of the 7th conference on File and storage technologies*, FAST '09, pages 85–98, Berkeley, CA, USA, 2009. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1525908.1525915`. 1.7, 3.1, 3.4.5, 3.5, 3.5

[37] Ajay Gulati, Chethan Kumar, Irfan Ahmad, and Karan Kumar. Basil: Automated io load balancing across storage devices. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 13–13, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?`

id=1855511.1855524. 1.7, 1.8, 5.4.7, 5.5

[38] Ajay Gulati, Arif Merchant, and Peter J. Varman. mclock: handling throughput variability for hypervisor io scheduling. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–7, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924974. 1.7, 3.1, 3.4.5, 3.5

[39] Ajay Gulati, Ganesha Shanmuganathan, Irfan Ahmad, Carl Waldspurger, and Mustafa Uysal. Pesto: Online storage performance management in virtualized datacenters. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 19:1–19:14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0976-9. doi: 10.1145/2038916.2038935. URL http://doi.acm.org/10.1145/2038916.2038935. 1.7, 1.8, 5.4.7, 5.5

[40] Daniel P Heyman and David Lucantoni. Modeling multiple ip traffic streams with rate limits. *Networking, IEEE/ACM Transactions on*, 11(6):948–958, 2003. 4.2.4

[41] Sadeka Islam, Srikumar Venugopal, and Anna Liu. Evaluating the impact of fine-scale burstiness on cloud elasticity. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 250–261, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3651-2. doi: 10.1145/2806777.2806846. URL http://doi.acm.org/10.1145/2806777.2806846. 4.1

[42] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. In *ACM SIGCOMM*, pages 219–230, 2013. 4.1, 4.5, 4.5

[43] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable message latency in the cloud. In *ACM SIGCOMM*, pages 435–448. ACM, 2015. 1.7, 3.6, 4, 4.1, 4.1, 4.1, 4.3.1, 4.5, 4.5, 4.5, 5.1, 5.1, 5.2.1, 5.3.1, 5.4.7, 5.5, 5.5, 6.1.2

[44] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. Response time service level agreements for cloud-hosted web applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 315–328, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3651-2. doi: 10.1145/2806777.2806842. URL http://doi.acm.org/10.1145/2806777.2806842. 4.5

[45] Wei Jin, Jeffrey S. Chase, and Jasleen Kaur. Interposed proportional sharing for a storage service utility. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '04/Performance '04, pages 37–48, New York, NY, USA, 2004. ACM. ISBN 1-58113-873-3. doi: 10.1145/1005686.1005694. URL http://doi.acm.org/10.1145/1005686.1005694. 1.7, 3.1, 3.4.5, 3.5

[46] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu. Triage: Performance differentiation for storage systems using adaptive control. *Trans. Storage*, 1(4):457–480, November 2005. ISSN 1553-3077. doi: 10.1145/1111609.1111612. URL `http://doi.acm.org/10.1145/1111609.1111612`. 1.7, 3.1, 3.4.5, 3.5

[47] Swaroop Kavalanekar, Bruce L. Worthington, Qi Zhang, and Vishal Sharda. Characterization of storage workload traces from production windows servers. In David Christie, Alan Lee, Onur Mutlu, and Benjamin G. Zorn, editors, *IISWC*, pages 119–128. IEEE, 2008. ISBN 978-1-4244-2778-9. URL `http://dx.doi.org/10.1109/IISWC.2008.4636097`. 2.1, 2.3.1, 3.1, 3.3.2, 3.4, 4.3.2, 4.4.6, 5.3.2

[48] Jim Kurose. On computing per-session performance bounds in high-speed multi-hop computer networks. In *ACM SIGMETRICS*, 1992. 1.7, 4.5

[49] Jean-Yves Le Boudec. Application of network calculus to guaranteed service networks. *IEEE Transactions on Information Theory*, 44(3):1087–1096, 1998. 4.5

[50] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, Berlin, Heidelberg, 2001. ISBN 3-540-42184-X. 1.8, 3.5

[51] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*, volume 2050. Springer Science & Business Media, 2001. 4.3.1, 5.1, 5.3.1, 5.4.7, 5.5, A.5

[52] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. Pslo: Enforcing the xth percentile latency and throughput slos for consolidated vm storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 28:1–28:14, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4240-7. doi: 10.1145/2901318.2901330. URL `http://doi.acm.org/10.1145/2901318.2901330`. 4.5

[53] Jörg Liebeherr, Dallas E Wrege, and Domenico Ferrari. Exact admission control for networks with a bounded delay service. *IEEE/ACM Transactions on Networking (TON)*, 4(6):885–901, 1996. 4.5

[54] Jörg Liebeherr, Yashar Ghiassi-Farrokhfal, and Almut Burchard. On the impact of link scheduling on end-to-end delays in large networks. *IEEE JSAC*, 29(5):1009–1020, 2011. 4.5, A.3

[55] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez. Façade: Virtual storage devices with performance guarantees. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 131–144, Berkeley, CA, USA, 2003. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1090694.1090710`. 1.7, 3.1, 3.4.5, 3.5

[56] Arif Merchant, Mustafa Uysal, Pradeep Padala, Xiaoyun Zhu, Sharad Singhal, and Kang Shin. Maestro: quality-of-service in large disk arrays. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, pages 245–254, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0607-2. doi: 10.1145/1998582.1998638. URL `http://doi.acm.org/10.1145/1998582.1998638`. 1.7, 3.1, 3.4.5, 3.5

[57] Jeffrey C Mogul and Ramana Rao Kompella. Inferring the network latency requirements of cloud tenants. In *Usenix HotOS XV*, 2015. 4.1

[58] Guy Nason. A test for second-order stationarity and approximate confidence intervals for localized autocovariances for locally stationary time series. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 75(5):879–904, 2013. 4.3.2

[59] Abhay K Parekh and Robert G Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993. 4.3.1

[60] Abhay K Parekh and Robert G Gallagher. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM Transactions on Networking*, 2(2):137–150, 1994. 4.3.1

[61] Nohhyun Park, Irfan Ahmad, and David J. Lilja. Romano: Autonomous storage management using performance prediction in multi-tenant datacenters. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 21:1–21:14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1761-0. doi: 10.1145/2391229.2391250. URL `http://doi.acm.org/10.1145/2391229.2391250`. 1.7, 1.8, 5.4.7, 5.5

[62] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized zero-queue datacenter network. In *ACM SIGCOMM*, pages 307–318, 2014. 4.1, 4.1, 4.5, 4.5, 5.1

[63] Felix Poloczek and Florin Ciucu. Scheduling analysis with martingales. *Performance Evaluation*, 79:56–72, 2014. 1.7, 4.5

[64] Jing-yu Qiu and Edward W Knightly. Inter-class resource sharing using statistical service envelopes. In *IEEE INFOCOM*, volume 3, pages 1404–1411, 1999. 1.7, 4.5

[65] Jean-Luc Scharbarg, Frédéric Ridouard, and Christian Fraboul. A probabilistic analysis of end-to-end delays on an afdx avionic network. *IEEE Transactions on Industrial Informatics*, 5(1):38–49, 2009. 4.5

[66] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: A cautionary tale. In *Proceedings of the 3rd Conference on Networked Systems*

Design & Implementation - Volume 3, NSDI'06, pages 18–18, Berkeley, CA, USA, 2006. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267680.1267698. 3.3.2, 4.3.3

[67] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 349–362, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387914. 1.7, 3.1, 3.4.5, 3.5, 3.5

[68] Aameek Singh, Madhukar Korupolu, and Dushmanta Mohapatra. Server-storage virtualization: Integration and load balancing in data centers. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 53:1–53:12, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9. URL http://dl.acm.org/citation.cfm?id=1413370.1413424. 1.7, 1.8, 5.4.7, 5.5

[69] David Starobinski and Moshe Sidi. Stochastically bounded burstiness for communication networks. In *IEEE INFOCOM*, volume 1, pages 36–42, 1999. 1.7, 4.5

[70] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *USENIX NSDI*, 2015. 4.1, 4.5, 4.5

[71] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A Software-defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522723. URL http://doi.acm.org/10.1145/2517349.2522723. 2.1, 3.5

[72] Guillaume Urvoy-Keller, Gérard Hébuterne, and Yves Dallery. Traffic engineering in a multipoint-to-point network. *IEEE JSAC*, 20(4):834–849, 2002. 4.5

[73] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter tcp (d2tcp). In *ACM SIGCOMM*, pages 115–126, 2012. 4.1, 4.5, 4.5, 5.1

[74] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Low latency via redundancy. In *ACM CoNEXT*, pages 283–294, 2013. 4.1, 4.5, 4.5

[75] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: performance insulation for shared storage servers. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, FAST '07, pages 5–5, Berkeley, CA, USA, 2007. USENIX Association. URL http://dl.acm.org/

`citation.cfm?id=1267903.1267908`. 1.7, 3.1, 3.4.5, 3.5

[76] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: enabling high-level slos on shared storage systems. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 14:1–14:14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1761-0. doi: 10.1145/2391229.2391243. URL `http://doi.acm.org/10.1145/2391229.2391243`. 1.7, 3.1, 3.3.1, 3.3.1, 3.4, 3.4.1, 3.4.2, 3.4.5, 3.5, 3.5, 4.5

[77] Kai Wang, Florin Ciucu, Chuang Lin, and Steven H Low. A stochastic power network calculus for integrating renewable energy sources into the power grid. *IEEE JSAC*, 30 (6):1037–1048, 2012. 4.5

[78] Shengquan Wang, Dong Xuan, Riccardo Bettati, and Wei Zhao. Providing absolute differentiated services for real-time applications in static-priority scheduling networks. *IEEE/ACM Transactions on Networking*, 12(2):326–339, 2004. 4.5

[79] Joel Wu and Scott A. Brandt. The design and implementation of aqua: an adaptive quality of service aware object-based storage device. In *Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 209–218, May 2006. 1.7, 3.1, 3.4.5, 3.5

[80] Yunjing Xu, Michael Bailey, Brian Noble, and Farnam Jahanian. Small is better: Avoiding latency traps in virtualized data centers. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 7:1–7:16, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2428-1. doi: 10.1145/2523616.2523620. URL `http://doi.acm.org/10.1145/2523616.2523620`. 3.5, 4.1

[81] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *USENIX NSDI*, pages 329–342, 2013. 1.7, 3.5, 4.1, 4.1, 4.5, 4.5, 5.1

[82] Opher Yaron and Moshe Sidi. Performance and stability of communication networks via robust exponential bounds. *IEEE/ACM Transactions on Networking*, 1(3):372–385, 1993. 1.7, 4.5

[83] Young Jin Yu, Dong In Shin, Hyeonsang Eom, and Heon Young Yeom. Ncq vs. i/o scheduler: Preventing unexpected misbehaviors. *ACM Trans. Storage*, 6(1):2:1–2:37, April 2010. 2.2.1

[84] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. Detail: reducing the flow completion time tail in datacenter networks. In *ACM SIGCOMM*, pages 139–150, 2012. 4.1, 4.5, 4.5

[85] Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel.

Storage performance virtualization via throughput and latency control. *Trans. Storage*, 2(3):283–308, August 2006. ISSN 1553-3077. doi: 10.1145/1168910.1168913. URL `http://doi.acm.org/10.1145/1168910.1168913`. 1.7, 3.4.5, 3.5, 5.1, 5.5

[86] Timothy Zhu, Alexey Tumanov, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *ACM SOCC*, pages 29:1–29:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3252-1. doi: 10.1145/2670979.2671008. URL `http://doi.acm.org/10.1145/2670979.2671008`. 1.8, 4, 4.1, 4.1, 4.3.1, 4.5, 4.5, 4.5, 5.1, 5.2.1, 5.12, 5.4.7, 5.5

[87] Timothy Zhu, Daniel S. Berger, and Mor Harchol-Balter. SNC-Meister: Admitting More Tenants with Tail Latency SLOs. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 374–387, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4525-5. doi: 10.1145/2987550.2987585. URL `http://doi.acm.org/10.1145/2987550.2987585`. 1.8, 5.1, 5.4.7, 5.5