1 Introduction

Meeting tail latency Service Level Objectives (SLOs) in shared networked storage systems is an important and challenging problem in datacenters. Our work is motivated by three trends:

First, companies like Google and Amazon are increasingly interested in long tails at the 99th and 99.9th percentile latencies [11, 12]. As technology improves, users are more accustomed to low latency and start to expect near instant response times. Furthermore, as workloads become more parallel, the need for low tail latencies becomes increasingly important since jobs often run at the speed of the slowest request.

Second, as workloads become increasingly data-driven, 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 can be completely avoided, storage latencies typically have the most impact on overall latency, particularly at the tail.

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

1.1 Thesis statement

In our proposed work, we will demonstrate that we can build a networked storage system that can meet tail latency SLOs for many workloads sharing the system. Our goal is to support hundreds of workloads in a shared cluster, each with its own tail latency SLO on storage requests traversing storage and network resources. Our solution will ensure that a workload’s requests complete within its tail latency SLO; in particular, we would like almost all requests (e.g., at least 99.9%) to complete within the specified SLO latency (e.g., 200ms). Thus, our SLOs are specified as a pair of numbers (SLO latency, SLO percentile) where the SLO latency specifies the desired latency, and the SLO percentile specifies the percentage of requests that should have a latency under the SLO latency.

We will study how to meet tail latency SLOs from the perspective of scheduling policies, admission control, and workload placement/migration. We will support real-world workloads, which are bursty both in their inter-arrival times and in their demands, running on multiple types of storage devices including magnetic disks, solid state drives (SSDs), and storage arrays. Within this context, we will answer the following questions:

Q1 (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)?

Q2 (traffic enforcement): How do we limit the impact of one workload on another?

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 different SLO percentiles (e.g., 90th, 99th, 99.9th percentile)?

Q5 (SLO-aware workload placement): When dealing with hundreds of workloads, which workloads should be co-located to best meet their SLOs?

Q6 (SLO-aware workload migration): Can we take advantage of migration to fix bad initial placements?

Q7 (heterogeneous storage): When working with a mix of heterogeneous storage devices (e.g., magnetic disk, SSD), where should a workload be run to meet tail latency SLOs?

Q8 (RAID arrays): How can our solutions apply to more complex RAID storage arrays?

Q9 (distributed storage): How can our solutions apply to distributed storage backends such as Ceph RADOS?

1.2 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 Sec. 8 for details).

Storage scheduling: Much of the prior work on storage scheduling is limited to the easier problem of sharing storage bandwidth [26, 33, 50, 55, 29, 59]. 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 work targets latency, but most of this work is focused on the aver-
age latency [42, 34, 41, 28]. Looking at the average can mask some of the worst-case behaviors that often lead to stragglers.

Recent works in the last couple years, Cake [56] and Avatar [62], 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 [62] is an Earliest Deadline First (EDF) scheduler with rate limiting support, aimed at meeting the 95th percentile. Avatar suffers from three limitations: (i) it does not address how to set rate limits, (ii) its rate limiting model is not configurable for workloads of varying burstiness, (iii), 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, we will 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 in the past two years, approaches for guaranteeing tail latency SLOs in datacenter networks [25, 32] and networked storage [63] have emerged. All of these works are based on analysis for the 100th percentile request latency, T100. T100 analysis is used despite the fact that typical users are only asking for SLOs at the 99th [60], or the 99.9th [12] percentiles. This is done for two reasons: First, the existing mathematics are easier to understand for a T100 (worst-case) latency bound compared to a T99 latency bound. Second, meeting a T100 SLO implies meeting tail latency SLOs at lower percentiles (e.g., 99th, 99.9th). However, admission decisions based on the T100 are far too conservative. The worst-case T100 analysis assumes all workloads behave adversarially where all workloads have bursts at exactly the same time.

The conservative nature of T100 admission control is known, and a new branch of theory called Stochastic Network Calculus (SNC) has been developed to address the shortcomings of T100 admission control [36, 6, 10, 48, 52, 61, 18, 7, 8, 16, 46]. However, all of these works are only in theory and have not been applied in practice to computer systems. In our SNC-Meister work, we will show how to 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 [27, 30, 45, 51, 14, 1, 2]. While load balancing works well for providing fairness or throughput SLOs, it does not work well in the presence of tail latency SLOs. Tail latency is not only affected by the load of each workload, but also the burstiness. We believe that a tail latency SLO-aware placement algorithm is needed for meeting tail latency SLOs.

The most relevant work to workload placement for tail latency SLOs is the recent work Silo [32]. Silo addresses the placement problem for VMs sharing a network; storage is outside the scope of their work. We plan to address the placement question for storage in our work.

1.3 Outline

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

System infrastructure: Sec. 2 describes the system infrastructure where we run our experiments. Our experiments run on top of commodity hardware, with storage exposed through Network File System (NFS). Similar to how IOFlow [53] enforces traffic, we build traffic enforcers for both storage and network. NFS storage traffic is enforced via a thin, transparent shim layer 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. Network prioritization and rate limiting is enforced 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 (aka TOS IP field), where priorities are marked in this field using Linux TC at the end hosts.

Sec. 2 also describes the real-world traces we use in our evaluation. Our traces come from the public SNIA dataset [35], which were collected from Microsoft production servers running applications such as Microsoft Exchange and LiveMaps. We find that real traces are very bursty, and this burstiness has a large impact on tail latency. Our solution will be designed to operate with bursty workloads.

PriorityMeister (Q1 & Q2): Sec. 3 describes our completed work, PriorityMeister [63]. To answer the first question, Q1, we compare multiple scheduling policies and demonstrate that it is possible to meet tail latency SLOs using a combination of priority and a new way of rate limiting, whereas other reactive policies do not cope well with the burstiness found in real workloads. Prioritization is our mechanism for providing 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 automatically selects storage and network priorities and rate limits based on each workload’s behaviors so as to meet each workload’s SLO.

The key tool that makes PriorityMeister work is a branch of theory called Deterministic Network Calculus (DNC) [38]. DNC allows us to calculate worst-case...
latex bounds based on the rate limits and priorities of the workloads sharing the system. These bounds then allow us to check if we have a mathematical guarantee of meeting each workload’s SLO. We use this DNC analysis as a tool for selecting the priorities for each workload.

PriorityMeister also answers Q2 by demonstrating how to set rate limits and priorities on both storage and network. We demonstrate that using multiple rate limits simultaneously for a single workload provides precise control over the workload’s behavior, which in turn more accurately limits the impact of one workload on another. We show that using multiple rate limiters per workload is practical by enforcing network rate limits using the Linux TC interface and storage rate limits via our shim layer on NFS RPCs.

**SNC-Meister (Q3 & Q4):** Sec. 4 describes our current work SNC-Meister, which tackles the admission control question, Q3. 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 admission control is necessary for providing SLO guarantees, and SNC-Meister is able to admit many more workloads than the state-of-the-art while meeting SLOs.

SNC-Meister is able to admit many more workloads because it is based on a new probabilistic theory called Stochastic Network Calculus (SNC) [17], which allows latency bound calculations at any percentile. By contrast, the state-of-the-art uses the worst-case DNC theory, which is too conservative in admitting workloads. The problem with the DNC theory is that it is a worst-case analysis technique that assumes adversarial behaviors in each of the workloads. While DNC is suitable for the 100th percentile, it is far too conservative for lower 99.9th or 99th percentiles, which are more common in literature.

SNC-Meister also answers Q4 by demonstrating how SNC can compute latency bounds at any desired percentile (e.g., 99th, 99.9th). Due to the probabilistic nature of SNC, we can simply plug in the SLO percentile as a parameter in the SNC analysis. We find that the ability to consider lower SLO percentiles is crucial to admission control, and our results so far indicate that SNC-Meister is able to admit 62% more workloads than approaches targeting the 100th percentile (e.g., DNC).

**Future work - workload placement (Q5, Q6, & Q7):** Sec. 5 describes our next planned work, which investigates the placement question, Q5. Most prior works treat the placement problem as a load balancing problem [27, 30, 45, 51, 14, 1, 2]. However, we believe that when there are latency SLOs, using the SLO in the placement decision will lead to better placements. For example, it is generally better to separate two workloads with low SLOs rather than to co-locate them. Our goal is to devise an SLO-aware placement algorithm that can scale to hundreds of workloads and servers.

In this work, we also plan to address the related migration problem, Q6. We see migration as an extension to the placement problem, where we additionally choose which workload to move and then place it using the placement algorithm. Our goal is to meet SLOs for more workloads while keeping the migration cost to a minimum.

We are also interested in the placement question in the context of heterogeneous storage with magnetic disks and SSDs, addressing Q7. We plan to design our placement algorithm so that it can also apply to heterogeneous environments.

**Future work - storage arrays (Q8 & Q9):** Sec. 6 describes our ideas on extending our work to storage arrays (e.g., RAID arrays, distributed storage), addressing Q8 and Q9. To get good performance, many applications stripe their data across many devices in a storage array. Striping reduces the load on each individual storage device by spreading the load across many devices. However, striping can sometimes lead to long delays due to hotspots, which makes it difficult to control for meeting tail latency SLOs. In this work, we will evaluate multiple ways of modeling and controlling traffic to storage arrays with a focus on distributed storage.

**Future work - if time permits:** Sec. 7 describes some open questions left to study if time permits.

One area of research that we have not yet considered is the question of pricing for tail latency SLOs. Since the ability to meet tail latency SLOs is based on both the workload’s behavior as well as its SLO, it is non-trivial to determine a price to charge for providing a workload a particular SLO. For example, even if two workloads have the same SLO, if one is bursty and another non-bursty, then the non-bursty workload should be charged less since its tail latency SLO is easier to satisfy.

Another interesting open question is how to extend our SNC-Meister work to support traffic shaping based on the probabilistic SNC arrival models. Probabilistic arrival models are much harder to enforce than deterministic arrival models since a strict enforcement of a probabilistic arrival model leads to significant queuing at the shaper. We believe that no perfect shaper exists, and that an approximation is needed.

We are also interested in how our work applies to tail latency of closed-loop workloads. In all of our work so far, we focus on tail latency for open-loop workloads because latency is most relevant for open-loop workloads. By contrast, throughput is most relevant for closed-loop workloads since the average latency is inversely proportional to the throughput by Little’s law. Nevertheless, there may be closed-loop workloads that are concerned about tail latency since the tail latency of closed-loop workloads is affected by other scheduling parameters.
Figure 1: Our system operates in a shared networked storage environment. Applications (squares) on client VMs access storage volumes (triangles) at servers over a network. Each workload comes with a tail latency SLO, e.g., 99% of workload $W_1$’s requests should have a latency less than 150ms ($T_{99} < 150\text{ms}$). 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.

Lastly, an open problem suggested by industry researchers is to investigate how to meet tail latency SLOs in the presence of a caching layer, such as a SSD cache. In this problem, we have the additional flexibility of choosing what data to cache as well as the ability to decide the amount of cache to allocate each workload. We believe that using the SLO can lead to better caching decisions for meeting tail latency SLOs for each workload.

Timeline: Sec. 9 presents a timeline of the proposed work.

2 System infrastructure

All of our work is designed to operate in shared networked storage systems as illustrated in Fig. 1. Applications (squares) on client VMs access storage volumes (triangles) on storage servers over a network. An application’s workload consists of a sequence of requests from a client to a server and back, where each request is represented by an arrival time, request type (read/write), request size (e.g., 4KB), and request offset. The end-to-end latency of a request is the total time between its arrival time and the completion time of the request, which includes all the queueing time at the client and server. Each workload comes with its own tail latency SLO, which is described by a SLO latency (e.g., 150ms) and a SLO percentile (e.g., 99%). For example, we write workload $W_1$’s SLO as $T_{99} < 150\text{ms}$, which represents a SLO where 99% of $W_1$’s requests have end-to-end latencies under 150ms.

Different workloads are allowed to have different SLOs, e.g., $W_2 : T_{99.9} < 200\text{ms}$ and $W_3 : T_{90} < 200\text{ms}$ and $W_4 : T_{99} < 400\text{ms}$.

The goal of our work is to meet these tail latency SLOs through a variety of techniques including prioritization, rate limiting, admission control, workload placement, etc. As shown in Fig. 1, 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). For example, the interference of workloads $W_3$ and $W_4$ with $W_1$ and $W_2$ can be minimized by assigning a higher priority to $W_1$ and $W_2$, which have the strictest latency requirements ($T_{99} < 150\text{ms}$, $T_{99.9} < 200\text{ms}$). Furthermore, using priority 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 receive a higher priority, and all other best effort workloads would operate at the default lowest priority. Lastly, to prevent starvation of low priority workloads, we also enforce rate limiting for each workload at the storage and network resources.

In our work, we implement storage prioritization and rate limiting on top of Network File System (NFS) running on commodity hardware. We insert a thin shim layer that intercepts and queues NFS RPC requests. Since NFS uses SunRPC, our shim layer hooks in via SunRPC and is entirely in userspace; no kernel modifications are necessary. For each workload, we create a first come first serve (FCFS) queue and track if the workload is within its rate limits. We execute requests from the highest priority non-empty queue that is within its rate limits.

To prioritize and rate limit network traffic, we use the Linux Traffic Control (TC) interface. On each end-host machine in our system, we use TC to configure PRIO queues for prioritization and Hierarchical Token Bucket (HTB) queues for rate limiting. We also use TC’s DS-MARK tagging to mark priorities within the Differenti-
When looking at the storage traces, we find that they are very bursty, and the bursts have varying durations and intensities. The first step in our work is to analyze each workload’s trace to build a mathematical model of this burstiness as well as the load of the workload. As part of the analysis, we use profiling data of the storage devices within our system. In our work, we implement profilers for measuring the performance characteristics of network and storage devices such as Microsoft Exchange, LiveMaps, and Ads servers.

Fig. 2 shows the flow chart of how we configure our system to meet tail latency SLOs. For a workload W, a user provides W’s desired SLO latency and percentile (e.g., 799 < 150ms) and a representative trace of W’s behavior. The trace can be captured as the workload runs, or the user can select a more representative trace from historical data. We require the following fields in a trace: arrival time, request type (e.g., read/write), request size (e.g., 4KB), and request offset. In our experimental results, we use traces from workloads running on Microsoft production servers [35]. The workloads come from applications such as Microsoft Exchange, LiveMaps, and Ads servers.

When looking at the storage traces, we find that they are very bursty, and the bursts have varying durations and intensities. The first step in our work is to analyze each workload’s trace to build a mathematical model of this burstiness as well as the load of the workload. As part of the analysis, we use profiling data of the storage devices within our system. In our work, we implement profilers for measuring the performance characteristics of network and storage devices such as Microsoft Exchange, LiveMaps, and Ads servers.

The results of our core component is a set of priorities and rate limits for each workload for network and storage along with the tail latency guarantees. At this point, we can optionally apply admission control by checking if the tail latency guarantees are less than the SLOs. If all SLOs are met, we admit the new workload and configure our system to enforce the priorities and rate limits for each workload. If not, we reject the new workload.

3 PriorityMeister

In this section, we describe our recently published work PriorityMeister [63], which compares multiple scheduling policies for meeting tail latency SLOs. We explore the question of how to arbitrate between multiple workloads sharing a networked storage system so that each workload can meet its SLO. We experiment with multiple approaches and develop a new approach for meeting tail latency SLOs.

The 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. We show that under bursty workloads, which are common in practice, reactive feedback-loop based 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. By contrast, PriorityMeister analyzes the burstiness and load of each workload and determines the set of priorities and rate limits for each workload that can meet tail latency SLOs.

PriorityMeister is different from prior approaches in that it uses a tail latency calculator that provides mathematical guarantees on 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 latency of each workload.
DNC is a general mathematical framework for calculating worst-case latencies (i.e., 100th percentile, T100). DNC can work with multiple competing workloads with different priorities and rate limits both in a single queue and networks of queues. In our paper [63], we demonstrate how to apply DNC to both storage and network resources. We also show how to deal with the end-to-end latency spanning both storage and network resources. The details of how to calculate the latency bounds for storage and network are described in the paper.

In Fig. 3, we compare multiple scheduling policies and show that PriorityMeister is the best policy for meeting tail latency SLOs at high percentiles. Our results show that priority can be a good scheduling mechanism for guaranteeing tail latency SLOs when priorities are intelligently configured based on our tail latency calculator. In this experiment, we run three co-located, latency-sensitive workloads based on real production traces along with a throughput-oriented workload. We show the 90th, 99th, and 99.9th percentile latencies of the three latency-sensitive workloads along with their SLOs as dashed lines in the corresponding color. The rightmost policy, equal-weight proportional sharing (ps), is a strawman example that shows that fair sharing is not the right solution for meeting tail latency SLOs. Cake [56] is the state-of-the-art reactive feedback-loop algorithm for meeting tail latency SLOs at the 99th percentile. However, it does not perform well in our experiments because we use real traces that are bursty, and reactive algorithms struggle to meet tail latency SLOs with bursty workloads. Earliest deadline first (EDF) and setting higher priorities to workloads with lower SLO latencies (bySLO) are both biased towards workloads with lower SLO latencies without considering each workload’s behavior. By not considering workload behaviors, some workloads violate SLOs at high percentiles (Fig. 3(b) and Fig. 3(c)). In this example, EDF and bySLO prioritize Workload C, and its higher load and burstiness causes SLO violations in Workload B (and sometimes Workload A). By contrast, PriorityMeister considers both workload behaviors and SLOs in determining priorities, which leads to meeting SLOs at even higher percentiles. PriorityMeister determines that it is better to prioritize Workloads A and B over C, which allows all SLOs to be met in this example. This is possible due to the tail latency calculator and a prioritizer algorithm we develop for efficiently determining priorities for meeting SLOs based on the tail latency calculator.

For the tail latency calculator to produce tight bounds on latency, we need to accurately characterize workload behavior. In our work, we find that workload behavior can be accurately characterized based on rate limits on the workload. 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 us to more accurately limit the effect of one workload on another. We show an example motivating this idea in Fig. 4, 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 (based on request size) 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.

Fig. 4 shows an example motivating the idea of multiple rate limiters on a high priority workload $H$. Note that there are many rate limit parameters $(r, b)$ that are sufficiently high to allow a workload to proceed without any queueing. Fig. 4(a) shows an example of the rate limit parameters (for one of the production server traces) where all of the points in the shaded region allows workload $H$ to proceed without queueing. In our work, we use this shaded region as a characterization of workload $H$’s behavior. Now to investigate how the choice of $H$’s rate limit affects performance, we try each of the rate limits marked by colored X’s and show the effect on lower priority workloads. Fig. 4(b) shows the 99.9th percentile latency of a medium priority workload $M$, and Fig. 4(c) shows the 99.9th percentile latency of a low priority workload $L$. If we select the (low rate, large bucket) rate limit for the high priority workload $H$ (green X in Fig. 4(a)), then the medium priority workload $M$ exceeds its SLO since it is delayed by large bursts of workload $H$ (green bar in Fig. 4(b) is above horizontal dashed SLO line). If we select the (medium rate, medium bucket) or (high rate, small bucket) rate limit for $H$, then the low priority workload $L$ exceeds its SLO since there is insufficient bandwidth leftover once $H$ consumes a medium or high rate. Thus, none of the rate limits individually allows us to meet the SLOs for both $M$ and $L$. So in PriorityMeister, we instead select multiple rate limits (all 3 X’s) simultaneously for $H$, which allows both $M$ and $L$ to meet their SLOs (blue bar). Using multiple rate limits simultaneously allows us to more accurately characterize and constrain $H$ without delaying $H$. This in turn helps $M$ and $L$ meet their SLOs.
Figure 6: Results when running workloads with a mixture of T99.9 SLOs at 150ms, 200ms, and 400ms. The left graph shows the number of admitted workloads under the No Admission Control, T100 Admission Control, 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 T100 Admission Control, 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 T100 Admission Control.

Figure 7: Same experiment as in Fig. 6 except with T90 SLOs. Even for a relatively low 90th percentile, it is still possible to exceed SLOs with No Admission Control. T100 Admission Control admits the exact same workloads as in Fig. 6 since it cannot distinguish between different SLO percentiles. SNC-Meister admits twice as many workloads as T100 Admission Control while still meeting SLOs.

4 SNC-Meister

In our current SNC-Meister work, we focus on the problem of 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.

Traditionally, admission control has primarily focused on the bandwidth of the workloads. By limiting the total bandwidth consumed by the workloads, high load and overload conditions can be avoided. However, when dealing with tail latency, it is insufficient to look at the load alone. We also need to consider the burstiness of the workloads.

While our prior work PriorityMeister does not discuss admission control, the DNC analysis used in PriorityMeister could easily be used for admission control. However, DNC is a worst-case queueing analysis targeting the 100th percentile latency, T100. Thus, PriorityMeister is suitable for admission control for T100 SLOs, but not for lower SLO percentiles such as the 99.9th and 99th percentiles. As we’ll see in our results, PriorityMeister [63] and other T100 Admission Control systems [32, 25] are too conservative in admitting workloads with lower SLO percentiles. This is because T100 guarantees need to cover 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, such as the T99.9, do not need to cover these unrealistic worst case scenarios.
The key difference in SNC-Meister is that it uses a new probabilistic analysis technique called Stochastic Network Calculus (SNC). SNC is designed to address some of the shortcomings of DNC by providing a mathematical framework for analyzing any latency percentile (e.g., 99%). So far, SNC has only been studied in theory, and SNC-Meister is the first computer system to apply this new branch of theory.

Fig. 5 shows a summary of our results comparing SNC-Meister, T100 Admission Control, and No Admission Control. We run 16 experiments, each with 35 workloads based on traces from Microsoft production servers [35]. We see that SNC-Meister is able to admit 62% more workloads than the state-of-the-art, T100 Admission Control. 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, and SNC-Meister admits many more workloads while guaranteeing SLOs due to its probabilistic SNC analysis.

Fig. 6 and Fig. 7 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. T100 Admission Control meets all SLOs, but only admits 29% of the workloads. Since T100 Admission Control cannot distinguish between a T99.9 and a T90 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 Fig. 8, 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 T100 Admission Control and find that SNC-Meister admits 54% more workloads with T99.9 SLOs (Fig. 8(a)) and 67% more workloads with T90 SLOs (Fig. 8(b)). Furthermore, these histograms have a clear separation between SNC-Meister and T100 Admission Control, which indicates that SNC-Meister admits more workloads than T100 Admission Control in almost all cases.

5 Future work - workload placement

Our work thus far assumes that each workload’s client and server locations are known and fixed. In the next phase of our research, we consider new storage volumes arriving over time, and we answer the question of which server we should place the new workload data on so as to best meet tail latency SLOs. Fig. 9 summarizes the placement questions we’re planning to address. We believe that the extra dimension of flexibility in determining workload placement will allow more workloads to be able to meet tail latency SLOs.

5.1 Preliminary results

There are multiple policies for choosing workload placement. For example, one could balance the number of workloads on each server, or one could alternatively try to balance the load on each server. The key idea in this work is that the placement decision should be based on not only the number and load of the workloads, but also the SLO of the workloads. In particular, we want to separate the workloads with tight SLOs as much as possible so that the workloads with tight SLOs don’t compete with each other. Note that our solution depends on workloads being prioritized; if all workloads have the same priority, it would be better to co-locate the workloads with similar SLOs while limiting the number of co-located workloads based on the SLO.

As an initial proof of concept, we run a small experiment comparing four placement policies for when a new workload arrives:

- **Random** - select random server to host the workload
Figure 9: A summary of the placement questions we are considering to explore.

**Workload placement:** when a new storage volume is added (e.g., purple triangle), which storage server should host the data? We will start in the homogeneous storage case and then move to the heterogeneous case with mixtures of SSDs and magnetic disks.

**Workload migration:** which storage volumes should be moved (e.g., green triangle) to allow meeting more SLOs?

**Client VM migration:** which client VMs should be moved (e.g., red square) to avoid client network congestion?

- **NumberBalanced** - select server with fewest number of workloads
- **LoadBalanced** - select server with lowest load
- **SLO-Balanced** - select server to guarantee SLOs based on PriorityMeister’s latency analysis

We experiment with 21 workload traces from Microsoft production servers running services such as Microsoft Exchange and LiveMaps [35]. We target 7 servers with NFS-mounted magnetic disks over a 1Gbps network. In our experiment, we randomize the ordering of the 21 workloads and place the workloads onto the servers one-by-one according to each of the placement policies. We then configure priorities in order of SLO, where the tightest SLO gets the highest priority. We next run PriorityMeister’s network calculus latency analysis and reject the workloads where SLOs cannot be guaranteed.

Fig. 10 reports the summary of the preliminary results, with this experiment repeated 800 times with different random orderings (yielding a total of 16800 placements). We run this experiment for three fixed sets of SLOs, where the tight SLOs range from 0.5s to 2.7s and the loose SLOs range from 0.5s to 4.4s. We find that the Random policy does not work well at all, as expected. We see that the NumberBalanced and LoadBalanced policies perform similarly since most of the 21 workloads have a roughly similar load. If the load between workloads were very different, we believe LoadBalanced would perform better than NumberBalanced. The best policy we’ve seen so far is the SLO-Balanced policy that also takes into account the SLO of each workload. We see the greatest benefit in the tight SLOs case where it’s hardest to guarantee the SLOs, and thus where placement matters most. Our preliminary results shows that SLO-Balanced rejects 50% fewer workloads than NumberBalanced and LoadBalanced in the tight SLOs case.

Fig. 11 shows a more detailed histogram view of the 800 experiments in the tight SLOs case. We see that in 10% of the 800 cases, SLO-Balanced is able to guarantee all workload SLOs (0 rejected), which is far greater than the other policies. Furthermore, we see that in nearly half of the cases, SLO-Balanced only needs to reject one workload, whereas the other policies reject many more workloads. This indicates that SLO-Balanced performs better than the other policies in most of the cases. However, SLO-Balanced is not perfect as it still rejects some workloads. We believe that there is room for improvement since there exists arrival orderings in which all SLOs can be guaranteed.

### 5.2 Next steps

There are many further questions we plan to investigate as a part of this workload placement project. This section lists in order the next steps we plan to take. First, we plan to run experiments to measure the tail latency of the workloads on real hardware. Second, we plan to explore other placement policies to further improve our results. One idea we’re interested in trying is to pack workloads while guaranteeing SLOs instead of balancing workloads across servers. By packing the workloads, we leave more empty servers that can potentially host high load or very bursty workloads in isolation. Another idea is to apply some heuristics to separate the workloads with tight SLOs.

Third, we plan to test how well our placement policies work as we increase the number of servers. Fourth, we will evaluate the scalability of the placement policy
Figure 10: A summary of the preliminary workload placement results. For each placement policy (described in Sec. 5.1), we place 21 workloads one-by-one onto 7 servers. We run PriorityMeister’s latency analysis and reject the workloads where SLOs cannot be guaranteed. We repeat this process for 800 random arrival orderings, yielding a total of 16800 placements, and plot the total number of rejected workloads on the y-axis. We run this experiment for three fixed sets of workload SLOs (x-axis), where the tight SLOs case has SLO latencies ranging from 0.5s to 2.7s and the loose SLOs case ranges from 0.5s to 4.4s. We see that the SLO-Balanced policy works best, particularly in the tight SLOs case where careful SLO-aware placement matters most.

Figure 11: A detailed histogram view of the 800 experiments in the tight SLOs case from Fig. 10.

6 Future work - storage arrays

Thus far, we have focused on simple storage systems (e.g., magnetic disk, SSD). Many applications today, however, use more complex storage systems with data striped across multiple storage devices in a storage array (e.g., RAID array, distributed storage). Striping is in many ways necessary for reducing the per-device load by spreading the load across many storage devices. However, striping presents new challenges in controlling and analyzing tail latency.

**RAID arrays:** The first step in this work is to study the performance characteristics of RAID arrays. We will test our existing magnetic disk and SSD storage profilers and build a new storage profiler designed for RAID arrays. We will also experiment with multiple techniques for controlling the storage traffic. We believe that one of the main concerns will be controlling the number of concurrent requests to the storage array. To effectively utilize all of the storage devices within the array, we need many concurrent requests. However, having too many concurrent requests could lead to long delays due to hotspots.

**Distributed storage:** The second step in this work is to study the performance characteristics of distributed storage arrays. We will target a block storage setting like Amazon EBS storage or OpenStack’s block storage running on a distributed storage backend such as Ceph RADOS Block Device. Each workload attaches to a storage volume, which is striped across the storage servers. We want to extend our work on meeting tail latency SLOs to this distributed setting. We plan to implement enforcement modules that can prioritize and rate limit traffic in a popular distributed storage backend such as Ceph RADOS.

One initial idea for handling distributed storage is to apply some of the ideas from controlling and profiling RAID arrays to distributed storage arrays. We are inter-
ested to see to what degree a distributed storage system can be treated as a large black box.

Another idea is to more finely control the traffic flows within an application to each of the storage servers. Since our analysis tools are only designed to handle flows from a single client VM to a single storage server, our initial plan is to decompose an application into multiple subflows, where each subflow represents the traffic between a single client VM and a single storage server. Working with the individual subflows makes it easier to apply some of our existing ideas, but the subflows may be too bursty relative to the application’s burstiness. We plan to compare this individual subflow approach to the aggregate black box approach.

Lastly, we are interested in evaluating whether it is beneficial to stripe data across a smaller subset of the servers. By striping across smaller subsets, we can avoid co-locating workloads with tight SLOs. While striping across more servers is better for throughput, we believe there may be a case for smaller striping for guaranteeing tail latency SLOs. We plan to start with some small proof-of-concept experiments, and then tackle more complicated questions such as striping size and placement.

7 Future work - if time permits

In this section, we describe multiple open problems we know exist and would like to solve, but anticipate that we won’t have time to research. These problems are likely to end up in my research statement when applying for jobs.

Tail latency SLO pricing: An important open problem for marketing our work as a cloud product is how to set prices for tail latency SLOs. There are multiple requirements for this problem. First, the price needs to simultaneously account for the SLO latency, SLO percentile, and the rate limits associated with the workload. Second, the pricing scheme must incentivize users to provide looser SLOs and tighter rate limits. Third, the pricing needs to be simple to understand, and the accounting must be practical to implement.

There are two main challenges that make latency SLO pricing challenging: latency is not a linear metric, and latency is workload specific. Since the effect on latency when multiplexing multiple workloads is non-linear, it is not obvious how to set prices based on the percentage of resources utilized. In particular, it becomes increasingly difficult (and sometimes impossible) to satisfy lower SLO latencies, so the price cannot be a linear function of the SLO latency. Furthermore, since latency depends on the burstiness and load of each workload, the pricing needs to account for workload burstiness (as constrained by the workload’s rate limits). Even if a workload has a low load, it may be difficult to guarantee a tail latency SLO if the workload is bursty, and a pricing scheme must capture this difficulty.

One possible direction for taking this research is to consider the notion of effective bandwidth as an intermediate metric of resource consumption. Effective bandwidth is the minimum throughput guarantee for a given workload to meet its latency SLO. Using the effective bandwidth allows us to translate the complexities of latency and workload rate limits into a single number that can be linearly priced. However, this effective bandwidth notion is not perfect as there is no incentive to provide tighter workload rate limits, which can significantly limit the number of workloads sharing the system.

Stochastic traffic shaping: In our SNC-Meister work, an open question that we did not get a chance to investigate is how to support traffic shaping based on probabilistic SNC arrival models. Our SNC-Meister work assumes that workloads behave similarly to their arrival model, and we show that our model is robust to some differences in workload behavior. However, the current SNC-Meister work does not guard against workloads that misbehave and flood the system with requests. Traffic shaping is necessary to prevent workloads from deviating too far from their expected behavior.

The main challenge in probabilistic traffic shaping is that the shaping can induce significant delays, which would affect the ability to meet tail latency SLOs. For example, the na"ıve approach is to simulate the desired arrival process and shape the traffic according to the simulated arrivals. However, the delay due to shaping is high, particularly for bursty workloads. Ideally, if a workload conforms to its desired arrival process, then there should be no delay due to traffic shaping. However, detecting a deviation from the desired shaping is difficult, and we are unaware of any existing theory for probabilistic traffic shaping.

Closed-loop workloads: In our work so far, the primary focus has been on tail latency for open-loop workloads since latency has the largest impact on open-loop workloads. However, another interesting question is how our work applies to tail latency of closed-loop workloads. While throughput is the primary metric for closed-loop workloads, there may be some closed-loop workloads (e.g., streaming apps, on-demand video transcoding) that are concerned about tail latency as well. We are interested in seeing how to apply our work to closed-loop workloads. We will also investigate the applicability of some theoretical results for closed-loop workloads such as distributional Little’s law.

SSD caching: An open problem suggested by industry researchers is to investigate how to meet tail latency SLOs in the presence of a caching layer, such as a SSD cache. In this problem, we have the flexibility of choosing what data to cache as well as the ability to decide the amount of cache to allocate each workload. For the purposes of meeting tail latency SLOs, we believe that
the SLO requirements can be used to make better caching decisions.

The primary challenge in this problem is managing the cache friendliness of workloads alongside their SLOs. For example, a cache friendly workload with a tight latency SLO might require a larger portion of the cache. On the other hand, a cache unfriendly streaming workload may not need the cache regardless of its SLO. Our primary goal in this problem is automatic cache management for meeting tail latency SLOs.

We believe that this SSD caching problem is a natural extension of the placement problem with heterogeneous storage. The primary difference would be in figuring out how to divide a workload’s requests into the cached and uncached requests. We hope that some techniques from solving the heterogeneous storage problem may carry over to this problem.

8 Detailed prior work

In this section, we describe in detail the related work to our research, spanning multiple problem domains including storage scheduling, admission control, and workload placement.

8.1 Storage Scheduling

Our work is different from prior storage scheduling work in two main ways. First, it is designed specifically for meeting tail latency SLOs in multi-tenant environments. Second, our work generalizes to multiple resources including network and storage.

Tail Latency: Most of the prior work on storage scheduling has focused on the easier problem of sharing storage throughput [26, 33, 50, 55, 29, 59]. Of the work that focuses on latency, most target the average latency [42, 34, 41, 28]. We are only aware of two storage schedulers, Cake [56] and Avatar [62], that consider tail latency behavior.

Cake [56] is a reactive feedback-control scheduler that adjusts weights associated with each workload to meet 99th percentile latency SLOs. When a workload’s SLO is not being met, more weight is given to the workload. When a workload’s SLO is sufficiently met, less weight is given to the workload. In our work, we take a different approach and overcome some of Cake’s limitations. Cake only handles one latency-sensitive workload with one throughput-oriented workload, whereas we handle multiple latency-sensitive and throughput-oriented workloads. Our work is designed to account for the burstiness found in real storage traces, and we can meet higher percentile latency SLOs (e.g., 99.9%), both of which are not possible using a reactive approach such as Cake.

While Cake addresses multiple resources for HBase (CPU) and HDFS (storage), it requires a mechanism to dynamically adjust weights for each resource. However, dynamically adjustable weights for networks is not readily available. One possible solution is to extend Cake to use network rate limits as a proxy for weights, but our tests show that it hurts performance more than it helps. In our work, we use priority, which is a simpler mechanism that is supported in many network switches.

Avatar [62] is an Earliest Deadline First (EDF) scheduler with rate limiting support. While Avatar looks at 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. One of the key results in our work addresses how to automatically determine rate limits based on workload traces, and we do this in a way that supports workloads of varying burstiness. Lastly, the focus in Avatar is solely on storage, and its EDF solution does not generalize to networks.

Multi-resource: A few recent papers investigate the challenges with multi-resource scheduling [53, 21, 20, 56, 26, 50]. Supporting latency SLOs across multiple resources is particularly challenging since the end-to-end latency is cumulative across all the resource stages (e.g., storage, CPU, network, etc.). One could imagine dividing the problem into meeting SLOs for each resource independently, but it is not obvious how to divide an end-to-end SLO into SLOs per resource stage. In our work, we build a single system that understands both storage and network and can automatically configure the system to meet end-to-end latency SLOs. Our multi-resource architecture is most similar to that of IOFlow [53]. IOFlow introduces a new software-defined storage architecture for both storage and network, 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.

8.2 Admission Control

Admission control is the problem of deciding upfront whether a new workload can be admitted into the system without causing SLO violations for itself or any other previously admitted workloads. Until recently, few works have discussed admission control in the context of providing tail latency guarantees [32, 63, 25]. There are two main approaches to providing tail latency guarantees. First, there is a body of work that makes admission decisions based on T100 (i.e., worst-case latency) guarantees. As shown in Sec. 4, T100 Admission Control is overly conservative, which leads to very few workloads being admitted. Second, there is a branch of theory called Stochastic Network Calculus (SNC) that provides tail latency guarantees at any percentile. However, all of this
SNC work is only in theory, and we are the first to apply this new branch of theory to practice in computer systems.

**T100 Admission Control:** There are three recent works that consider T100 latency guarantees. QJump [25] offers a T100 latency guarantee for low-throughput applications in datacenter networks. This guarantee is achieved by rate limiting applications and performing a worst-case T100 latency analysis. However, as noted by the authors of QJump, the rate limiting severely limits the per-application throughput. In order to offer a configurable trade-off between latency and throughput, QJump relaxes the latency guarantee with the use of a “throughput factor” $f$. While using a higher throughput factor $f$ allows for much higher throughput and network utilization, a tail latency guarantee is only achieved by the lowest throughput factor $f = 1$, in which case applications are severely rate limited. Therefore, QJump’s latency guarantees remain limited to T100 SLOs for low-throughput applications.

Silo [32] offers per-tenant T100 latency guarantees in multi-tenant datacenter networks. Silo automatically finds tenant placements where each tenant’s T100 SLO can be satisfied. The admission and placement decisions are obtained using Deterministic Network Calculus (DNC) [39], a branch of mathematics for analyzing T100 latency. Like QJump, Silo is only applicable to networks, whereas our work generalizes to end-to-end latency in both network and storage.

Our first work, PriorityMeister [63], is also based on T100 latency analysis using DNC. The DNC mathematics used in PriorityMeister are based on the tight analysis in [4], which are tighter than the equations in Silo. However, even with tight DNC analysis, T100 Admission Control approaches are conservative since they must assume adversarial worst-case conditions, which are unrealistic in many environments.

The DNC theory has a long history of applications for latency guarantees. Besides these three recent works, many old T100 Admission Control algorithms have been studied in the context of internet QoS [15, 43, 44, 40, 37, 54, 58]. They also face the same challenges of being overly conservative.

**Stochastic Network Calculus:** The modern SNC theory evolved as an extension of the DNC theory to capture statistical multiplexing gains and enable accurate guarantees for any percentile of the latency [36, 6, 10, 48, 52, 61, 18, 7, 8, 16, 46]. However, all of this work is in theory, and we are not aware of any computer systems using SNC in practice. The only practical applications so far of this relatively new SNC theory are in the modeling of critical infrastructures such as avionic networks [49] and the power grid [57, 19], which point to the robustness of SNC theory.

**Other Admission Control:** Besides T100 Admission Control and the SNC theory, there are some prior works that use live traffic measurements to make admission decisions [9, 13, 23, 22, 24, 31]. Unfortunately, these approaches were later found to frequently miss the desired performance targets [5], and subsequent work has met similar challenges [47].

### 8.3 Workload Placement

Most of the prior work on storage workload placement and migration focus on load balancing [27, 30, 45, 51, 14]. These works use techniques such as simulated annealing, hill-climbing, multi-dimensional bin-packing, and greedy heuristics to balance the load. Load balancing works well for providing fairness and avoiding overloaded servers. However, load balancing is not ideal for meeting tail latency SLOs because tail latency is not only affected by the load, but also the burstiness of each workload. We believe that different placement policies are required for meeting tail latency SLOs, and our preliminary results in Sec. 5 motivate further investigation.

There are multiple older HP papers [1, 2, 3] that tackle the more general problem of automatically provisioning storage systems, which encompasses the placement problem. These works present multiple algorithms for provisioning storage systems and assigning workload data to the storage devices. However, they are primarily focused on storage throughput, and their work is more similar to load balancing than meeting tail latency SLOs.

The most relevant work to workload placement for tail latency SLOs is the recent work Silo [32]. Silo introduces a VM placement policy for a set of VMs sharing a network. Silo uses first-fit heuristics to balance the load. Load balancing works well for providing latency guarantees on network packets, but storage is outside the scope of their work. In our work, we will address the placement question for storage while also factoring network latencies into account.

### 9 Timeline

**Jan-Mar 2016:**
- Finish SNC-Meister (Sec. 4)
- Work on workload placement project (Sec. 5)
  - scalability
  - placement algorithms

**Apr-Jun 2016:**
- Finish workload placement project (Sec. 5)
  - heterogeneous storage
  - migration

14
• Hope to submit to conference around May (e.g., OSDI or SOCC)
• Work on storage arrays project (Sec. 6)
  – develop storage model for RAID array
  – build enforcement modules for a distributed storage system (e.g., Ceph)

Jul-Sept 2016:
• Finish storage arrays project (Sec. 6), assuming no unforeseen problems. If issues arise, we may replace with a project from Sec. 7.
• Hope to submit to conference around Sept. (e.g., NSDI, FAST, EuroSys)

Oct-Dec 2016:
• Work on writing thesis
• Prepare job applications
• Prepare job talk
• Finish last teaching assistantship

Jan-Apr 2017:
• Finish thesis
• Job interviews
• Work on other future work, time permitting (Sec. 7)

May 2017:
• Graduate

References


