PriorityMeister: Tail Latency QoS for Shared Networked Storage

Timothy Zhu* Alexey Tumanov* Michael A. Kozuch†
Mor Harchol-Balter* Gregory R. Ganger*

Carnegie Mellon University*, Intel Labs†

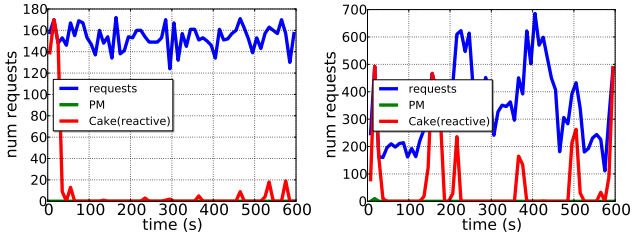
Abstract

Meeting service level objectives (SLOs) for tail latency is an important and challenging open problem in cloud computing infrastructures. The challenges are exacerbated by burstiness in the workloads. This paper describes PriorityMeister – a system that employs a combination of *per-workload priorities* and *rate limits* to provide tail latency QoS for shared networked storage, even with bursty workloads. PriorityMeister automatically and proactively configures workload priorities and rate limits across multiple stages (e.g., a shared storage stage followed by a shared network stage) to meet end-to-end tail latency SLOs. 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. PriorityMeister is also robust to mis-estimation of underlying storage device performance and contains the effect of misbehaving workloads.

1. Introduction

Providing for end-to-end tail latency QoS in a shared networked storage system is an important problem in cloud computing environments. Normally, one might look at tail latency at the 90th or 95th percentiles. However, increasingly, researchers and companies like Amazon and Google are starting to care about long tails at the 99th and 99.9th percentiles [4, 5, 24]. This paper addresses tail latencies even at the 99.9th and 99.99th percentiles.

Meeting tail latency 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.



(a) synthetic trace – low burstiness (b) real trace – high burstiness

Figure 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 colocated 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 [21]) is acceptable when there is little burstiness, but incurs many violations when bursts occur. PriorityMeister (PM) provides more robust QoS behavior. Details of the system setup, traces, algorithms, and configurations are described later in the paper. These graphs are provided up front only to illustrate the context and contribution of the new approach.

Much of the prior work on storage scheduling is limited to the easier problem of sharing storage bandwidth [9–11, 18, 20, 22]. Sharing bandwidth is easier than latency QoS because bandwidth is an average over time that is not affected by transient queueing. Some prior work targets latency QoS, but most of this work is focused on the average latency [8, 12, 15, 16]. Looking at the average can mask some of the worst-case behaviors that often lead to stragglers.

Recent work in the last couple years, Cake [21], 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. Fig. 1 illustrates this

SOCC '14, November 03 - 05 2014, Seattle, WA, USA.
Copyright 2014 ACM 978-1-4503-3252-1/14/11. . . $15.00.
http://dx.doi.org/10.1145/2670979.2671008

point; analysis and more experiments appear later. Fig. 1(a) shows that when the request rate (blue line) is not bursty, Cake meets latency SLOs with infrequent violations (red line). Fig. 1(b), on the other hand, shows that when the workload is much burstier 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 paper introduces PriorityMeister (PM), a 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 Sec. 3.2, a key idea in PriorityMeister is to use *multiple* rate limiters simultaneously for each workload at each stage. Using multiple rate limiters simultaneously allows us to better bound the burstiness of a workload. With a good bound on the burstiness for each workload, we build a model to estimate the worst-case per-workload latency. Our model is based on *network calculus*, an analysis framework for worst-case queueing estimation.

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. Our analytical model for latency allows PriorityMeister to quickly search 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 potentially allows more workloads to meet their SLOs.

PriorityMeister makes 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 latency SLOs. PriorityMeister achieves these goals by combining 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 [21]. 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 Fig. 5, Fig. 6, and Fig. 11), even when the bottleneck is at the network rather than storage (see Fig. 9). Third, we show that PriorityMeister is robust to mis-estimation in storage performance (see Fig. 10), varying degrees of workload burstiness (see Fig. 7), and workload misbehavior (see Fig. 8). Fourth, we come up with a *simple* approach as a starting point for PriorityMeister, which we call *bySLO*, and we find that it performs surprisingly well, also outperforming Cake (see Fig. 11).

Scheduler	Latency SLO	Multi-resource
Argon [20]	No	No
SFQ(D) [11]	No	No
AQuA [22]	No	No
mClock [10]	No	No
PARDA [9]	No	Yes
PISCES [18]	No	Yes
Maestro [16]	Average latency	No
Triage [12]	Average latency	No
Façade [15]	Average latency	No
pClock [8]	Average latency	No
Avatar [25]	95th percentile	No
Cake [21]	99th percentile	Yes
PriorityMeister	$>$ 99th percentile	Yes

Table 1. Comparison of storage schedulers.

2. Previous 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 1 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 [9–11, 18, 20, 22]. Of the ones that focus on latency, most of them target the average latency [8, 12, 15, 16]. We are only aware of two storage schedulers, Cake [21] and Avatar [25], that investigate tail latency behavior.

Cake [21] is a reactive feedback-control scheduler that adjusts proportional shares to meet 99th percentile latency SLOs. Our goals are similar, but we take a different approach and 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 [25] 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 [6, 7, 9, 18, 19, 21]. 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 [19]. 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 [24] 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 [1] addresses the problem of delays from long network switch queues by rate limiting and shifting the queueing to the end hosts. Xu et al. [23] 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 network calculus for modeling workloads and uses concepts such as arrival curves and service curves [14]. Our latency analysis is similar to a recent theory paper by Bouillard et al. [3].

3. Architecture

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 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, a request size, and a 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.

3.1 PriorityMeister system design

Fig. 2 shows the QoS components in our system in green. We have local QoS enforcement modules at each of the clients and servers to control access to storage and networking resources. Each enforcement module is independent and only needs to act upon QoS parameters received from our global controller. This allows PriorityMeister to take advantage of global information. Our system architecture is similar to that in IOFlow [19], which has been shown to scale and tolerate controller failure.

The two primary QoS parameters that PriorityMeister automatically configures for each enforcement module 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. As we will see in Sec. 3.2, PriorityMeister is unusual in that we deploy *multiple* rate limiters for each workload at each stage.

Our rate limiters are built upon a leaky token bucket model that is parameterized by a rate and a token bucket size. When a request arrives, tokens are added to the token bucket based on the request. For networks, the number of tokens added corresponds to the number of transmitted bytes for the request. For storage, the number of tokens added corresponds to the estimated amount of time for the storage device to process the request. The estimation is performed by the storage estimator (see Sec. 4.2). If there is space in the bucket to add tokens without exceeding the configured token bucket size, then the request is allowed to continue.

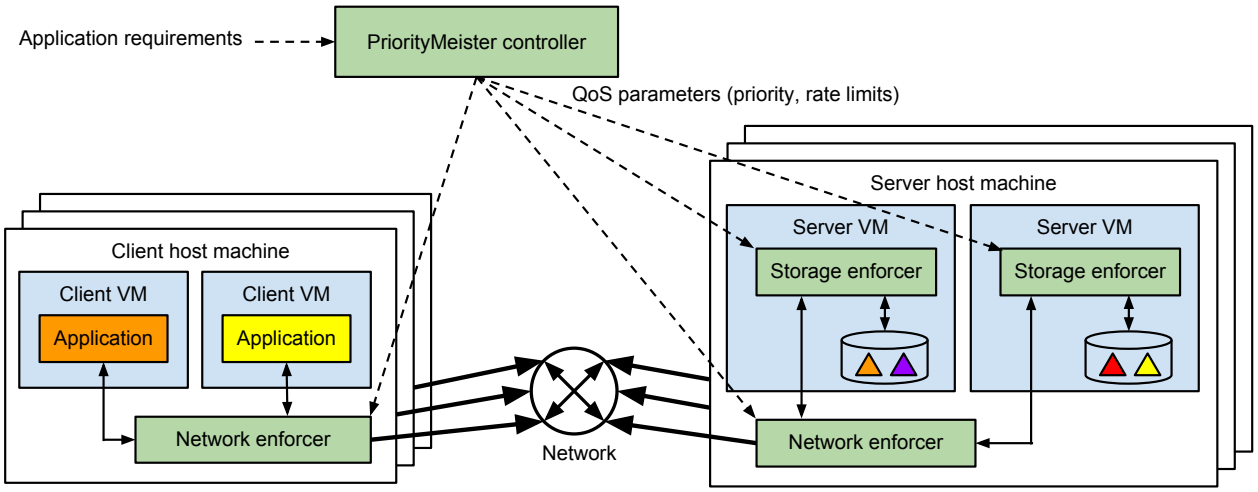


Figure 2. QoS components in PriorityMeister (green). The PriorityMeister global controller automatically configures our local storage and network enforcers running on each of the clients and servers based on application requirements. Multiple applications can run on the same client host in VMs. Applications access data hosted by storage servers through a shared network. A server host machine can contain multiple storage devices (cylinders) that can contain data (triangles) for different applications.

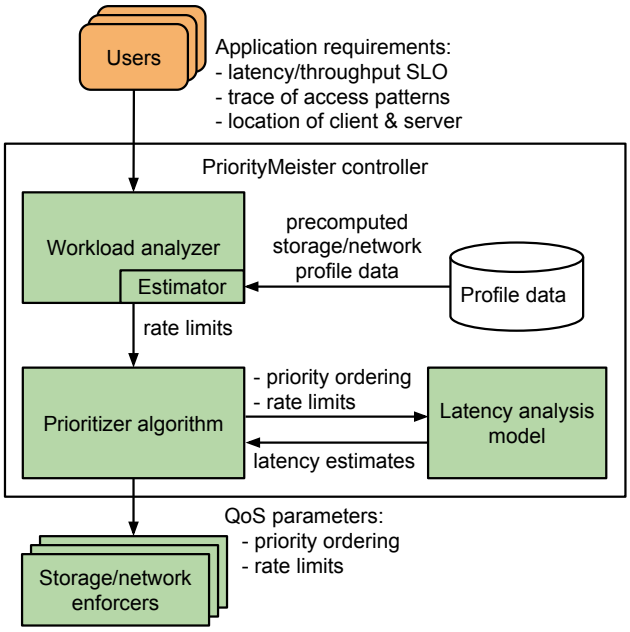


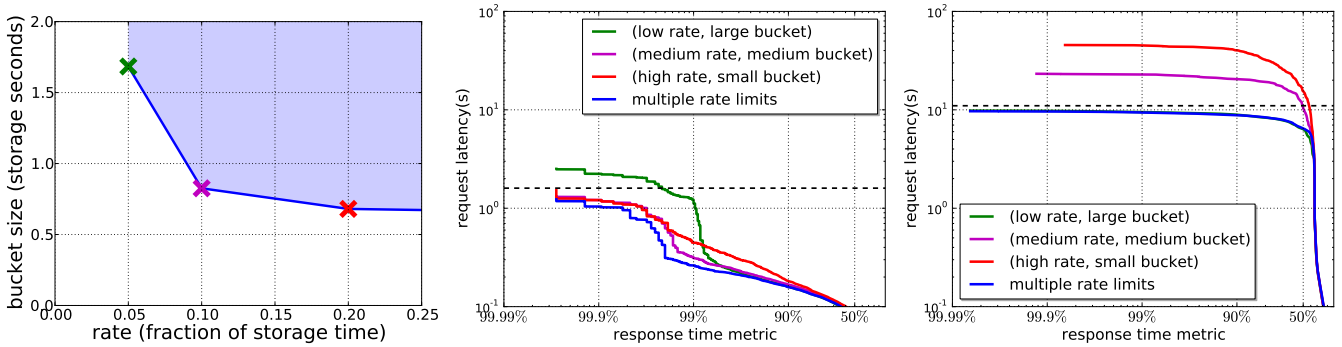
Figure 3. PriorityMeister controller dataflow diagram.

Otherwise, the request is queued until there is sufficient space. Space becomes available as tokens continuously leak from the bucket at the configured rate.

3.2 PriorityMeister controller design

Fig. 3 shows the design of our PriorityMeister controller. Users provide their goal, which can be a latency SLO or throughput SLO, a representative trace of application access patterns, and the location of their client and data (e.g., client & server IP addresses). The trace can be automatically captured as the application is running, or a power user can select a more representative set of access patterns using application-specific knowledge. A latency SLO is a value that specifies the maximum acceptable latency of a request. A throughput SLO is a value that specifies the total time in which the given trace of requests should complete.

We now discuss how PriorityMeister sets rate limits. We start with a thought experiment. Consider a workload, W, that needs to be high priority to meet its SLO. By giving W high priority, lower priority workloads will clearly have increased latency. Constraining the rate (r) and token bucket size (b) for workload W will limit the negative effect of W on lower priority workloads. Suppose we wish to have W run in an unfettered manner. There are many rate limiter (r,b) pairs for W that are high enough such that W is unaffected by them. W will only notice these (r,b) constraints if W "misbehaves" in the sense that it deviates markedly from its representative trace of access patterns. Fig. 4(a) depicts the set of (r,b) pairs that allow W to run in an unfettered manner, provided it doesn't misbehave. So from W's perspective, any of these (r,b) pairs are acceptable. However, lower priority workloads are affected differently by picking the green X (r,b) pair as opposed to the red X (r,b) pair. In Fig. 4(b), note that using larger token bucket sizes (e.g., green X) leads to higher tail latencies when W misbehaves. In Fig. 4(c), note that using higher rates (e.g., red X) causes the low priority workload to starve when W misbehaves. Using smaller bucket sizes and lower rates on W would help lower priority workloads, but such an (r,b) pair does not exist for W in Fig. 4(a). The key idea in PriorityMeister is to limit W by *all* (r,b) pairs in the shaded area of Fig. 4(a), which gives us the benefit of small bucket sizes and low rates. Specifically, we pick *multiple* (r,b) pairs along the border, such as the X's shown,



(a) rate limit pairs of high priority workload (b) latency of medium priority workload (c) latency of low priority workload

Figure 4. How token bucket parameters (rate, bucket size) limit the effect of a misbehaving high priority workload on lower priority workloads. In this experiment, we run 3 workloads together on the same disk. We cause the high priority workload to misbehave by inducing a large 40 sec burst of work in the middle of the trace. Fig. 4(a) shows the eligible set of (r,b) parameters for the high priority workload without the misbehavior (Workload B in Table 2). Fig. 4(b) and Fig. 4(c) show the effect, respectively, on the medium priority and low priority workload when the high priority workload misbehaves. Each colored X in Fig. 4(a) corresponds to a similarly colored line in Fig. 4(b) and Fig. 4(c). The similarly colored lines show the effect on the lower priority workloads when the misbehaving high priority workload is limited by the corresponding rate limit parameters. By 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 allows both lower priority workloads to meet their SLOs by using multiple rate limiters simultaneously (blue line).

and deploy these rate limiters simultaneously for W. That is, a request is queued until tokens can be added to *all* of its token buckets.

As illustrated in Fig. 3, PriorityMeister begins with our *workload analyzer*, which characterizes each workload by generating the workload's (r,b) pair region (e.g., Fig. 4(a)). Next, our *prioritizer algorithm* searches over the space of priority orderings to determine workload priorities that will best meet the workload SLOs. Our algorithm is built on a *latency analysis model*, which takes as input a priority ordering and set of rate limit (r,b) pairs and outputs worst-case bounds on latency for each workload based on network calculus. Our prioritizer algorithm does not search over all priority orderings; instead, we develop a simple greedy algorithm for picking "good" orderings (see Sec. 4.3). This provides an interesting alternative to having system administrators manually setting priorities, which is both costly and error prone. Users are inherently bad at choosing priorities since they may not be aware of the other workloads in the system. PriorityMeister chooses priorities automatically using high-level SLOs, which are much easier for a user to specify.

Lastly, the priorities and the rate limits are sent to the enforcement modules. Since we proactively analyze workload bursts, there is no need to run the prioritizer algorithm again until there is a change in the user requirements (e.g., change in application access patterns or SLO), or if new workloads arrive. When there is a change, we will rerun the prioritizer algorithm and potentially readjust priorities and rate limits.

4. Implementation

The dataflow within PriorityMeister is illustrated in Fig. 3 and is described in detail in this section.

4.1 Workload analysis

The role of workload analysis is to generate rate limits for each workload such that a workload is not hindered based on the workload's representative trace of access patterns. That is, we calculate the set of (rate, bucket size) pairs along the border (blue line) in Fig. 4(a). To calculate an (r,b) pair, we select a rate r and replay the workload's trace using an unbounded token bucket with the selected rate. We set the bucket size b to the maximum token bucket usage, which corresponds to the minimum token bucket size such that the workload is unhindered. We do this across a range of rates to get the entire set of (r,b) pairs.

4.2 Estimator

Estimators are used as part of workload analysis and in our storage enforcer to determine the number of tokens associated with a request. They provide a way of abstracting a variety of request types (e.g., read, write) and request sizes into a single common metric. PriorityMeister currently supports two types of estimators: storage and network. For storage, we use "work" as the common metric, measured in the units of time. Work denotes the time consumed by a request without the effects of queueing. For network, we use transmitted bytes as the common metric, which does not change across different link bandwidths.

Storage estimator: The storage estimator is responsible for estimating the amount of storage time consumed by

a request. The estimator does not need to be perfect, as demonstrated in Sec. 6.5. Rather, it is used for determining priority settings, which works effectively with approximate estimates.

Our storage estimator is similar to the table-based approach in [2]. We profile our storage devices a priori to build tables for (i) bandwidth and (ii) a base time to service a request. Our tables are parameterized by request type (e.g., read, write), request offset, and distance between subsequent request offsets (i.e., offset − previous offset). Additionally, we keep a history of requests to capture sequential accesses, and we assume a base time of 0 for sequential accesses. We then estimate the storage time to service a request as $\frac{\text{request size}}{\text{bandwidth}} + \text{base time}$.

Network estimator: The network estimator is responsible for estimating the number of bytes transmitted by a request. We use a simple estimator based on the request type and request size and find that it suffices. For read requests, there is a small request sent to the server and a large response back from the server. For write requests, there is a large request sent to the server and a small response back from the server.

4.3 Prioritizer algorithm

The prioritizer algorithm is responsible for finding a priority ordering that can meet the workload 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, 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 lowest latency, we assign w to be the lowest priority of the remaining set of workloads. We now repeat the process until all workloads have been assigned priorities at each stage.

4.4 Latency analysis model

The latency analysis model estimates worst-case latencies for the workloads under a given priority ordering and workload rate limits. The model we use in our system is based on the theory of network calculus. The main concepts in 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 will 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 is bytes for networks and storage time for storage. 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 $\alpha_w(t) = min_{i=1}^{m}(r_i * t + b_i)$ where workload w has m rate limit pairs (r_1, b_1), ..., (r_m, b_m). 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 [3], 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.

Rate limit constraints: We need to constrain the extent to which other workloads, k, can interfere with workload w. For a particular workload k, let (r_1, b_1), ..., (r_m, b_m) 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 (r_i, b_i). These constraints apply rate limits to each of the relevant time periods for workload k, and are added for each workload k.

Work conservation constraints: 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.

Objective function: 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 $t_{out}^{q_n} - t_{in}^{q_1}$.

4.5 Storage enforcer

Storage enforcement is responsible for scheduling requests at each of the storage devices. Our current implementation exposes storage as NFS mounts, and our storage enforcer is built as an interposition layer on top of NFS. Since NFS is based on SunRPC, we were able to hook into NFS at the RPC layer without needing to resort to kernel modification. Our storage enforcer creates queues for each workload and performs arbitration between the different workloads based on the priorities and rate limits assigned by the PriorityMeister global controller.

4.6 Network enforcer

Network enforcement 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. Since Hierarchical Token Bucket (HTB) queues can only support two rate limiters, we chain multiple HTB queues together to represent the multiple rate limiters for each workload. We then use DSMARK to tag the packets with DSCP flags (i.e., the TOS IP field), which instruct the network switch on how to prioritize packets. Our network switches support 7 levels of priority for each port, and using these priorities simply requires enabling DSCP on the switch. To get prioritization on the host as well as the switch, we lastly add a PRIO queue. To identify and route packets through the correct TC queues, we use filtering based on the source and destination addresses, which are different per VM.

4.7 Network reprioritization algorithm

Since networks can only support a limited number of priority levels, we also include an extra step after the prioritizer algorithm to adjust the priorities to fit within the supported priority levels. We sort the workloads as determined by the prioritizer and then partition the workloads among the number of supported priority levels. We determine the best partition points by running the latency analysis model and minimizing $\sum_{workload\ w} \frac{violation_w}{SLO_w}$.

5. 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.

5.1 Traces

We evaluate our system implementation using a collection of real production (described in [13]) and synthetic storage 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 2 provides a description of all 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 illustrated in [17], 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).

5.2 SLOs

Each workload in a given experiment has its own latency SLO, which is shown in the results section as a dashed horizontal 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 queueing time experienced by the client.

Not all requests in a workload will necessarily meet its SLO, so we also use the metric of a latency percentile to

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

Table 2. Workload traces used in our evaluation.

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.

5.3 Policies

In our experiments, we compare 5 QoS approaches: Proportional fair-share (ps), Cake [21], 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 [21]. 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 attempt to 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 also try another extension to 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 [6] 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 Sec. 6 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 simply assign workload priorities in order of the workload latency SLOs. That is, we assign the highest priority to the workload with the lowest SLO. This is supported for both network and storage. We find in Sec. 6.5 that this simple policy does surprisingly well, and we recommend bySLO as a simple way of getting started as a stand-in for our prioritizer algorithm.

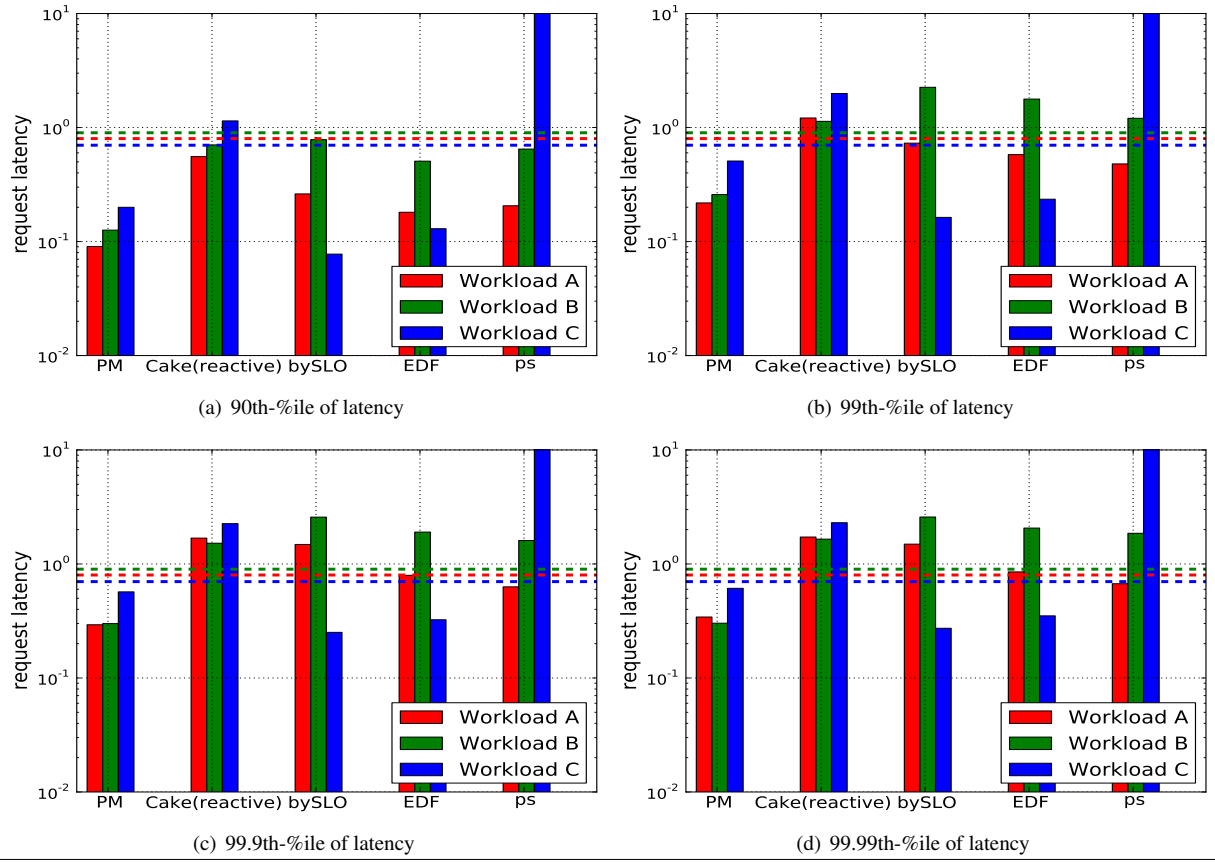
PriorityMeister (PM). PriorityMeister is our primary policy that we compare against the other policies. Its architecture (Sec. 3) and implementation (Sec. 4) are described earlier.

5.4 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.

6. Experimental Results

This section evaluates PriorityMeister (PM) in comparison to other state of the art policies across multiple dimen-



(a) 90th-%ile of latency

(b) 99th-%ile of latency

(c) 99.9th-%ile of latency

(d) 99.99th-%ile of latency

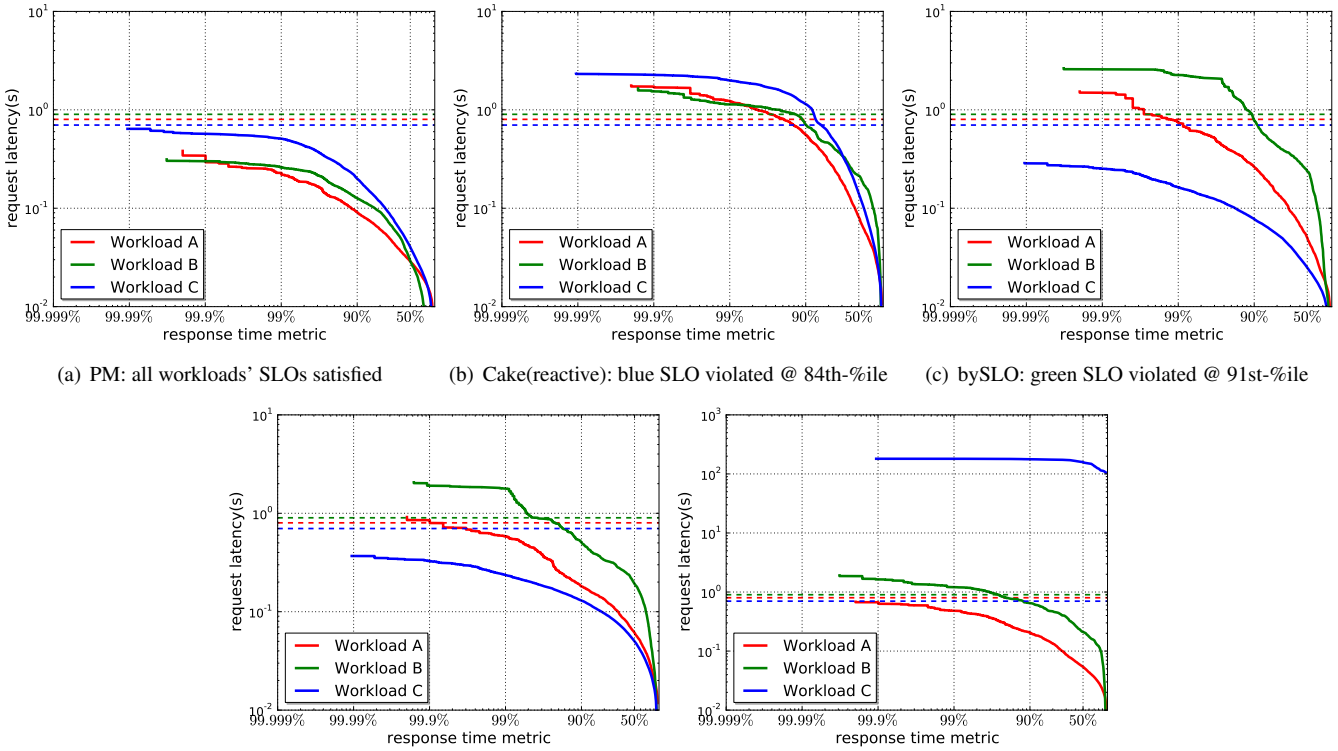
Figure 5. PriorityMeister (PM) is the only policy that satisfies all SLOs across all %iles. In this experiment, we replay three latency-sensitive workloads derived from production traces (Workloads A, B, and C, from Table 2) sharing a disk with with a throughput-oriented workload (Workload L; not shown) 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 Sec. 5.3).

sions. Sec. 6.1 demonstrates the ability of PriorityMeister in meeting tail latency SLOs on a set of production workload traces [13]. 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. Sec. 6.2 investigates the differences between proactive (PriorityMeister) and reactive (Cake [21]) 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. Sec. 6.3 then proceeds to show that PriorityMeister's prioritization techniques are safe to workload misbehavior through its automatic configuration of rate limits.

In Sec. 6.4, 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 Sec. 6.5. First, we show that PriorityMeister works with simple storage estimators and is robust to estimator inaccuracy. Second, we show that PriorityMeister can work under different SLO requirements. Surprisingly, we also find that the simpler bySLO policy, which we haven't seen in literature, does well in many cases. We believe that bySLO is a simple way of getting started, but there are still some cases where prioritizing by SLO does not work, as detected and corrected for by PriorityMeister.

6.1 PriorityMeister tail latency performance

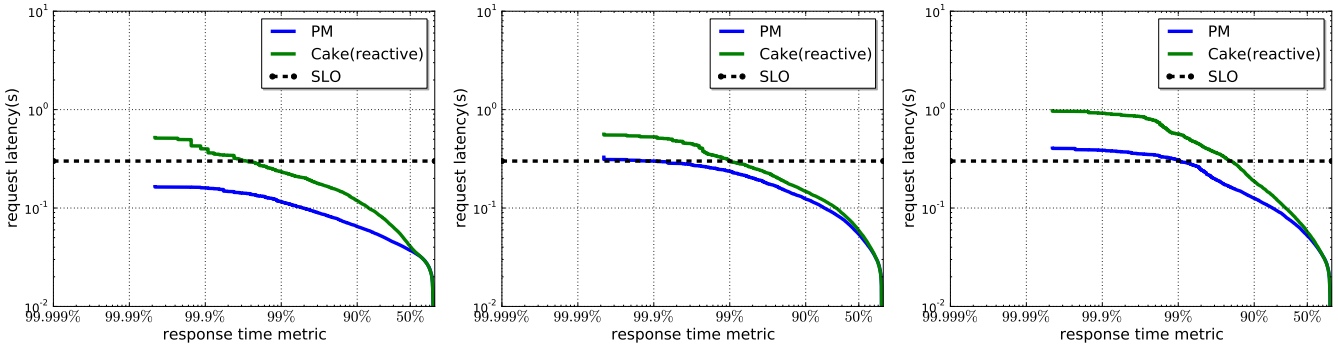
Fig. 5 plots tail latency performance across multiple policies, colocated workloads, and tail latency percentiles. PriorityMeister (PM) is the only policy that meets SLOs for all workloads across all percentiles. In this experiment, we show a typical example where the storage is a bottleneck. We replay 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.



(a) PM: all workloads' SLOs satisfied (b) Cake(reactive): blue SLO violated @ 84th-%ile (c) bySLO: green SLO violated @ 91st-%ile

(d) EDF: green SLO violated @ 97th-%ile (e) PS: blue SLO always violated

Figure 6. Request latency at different percentiles for each policy. Same experiment as in Fig. 5 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).

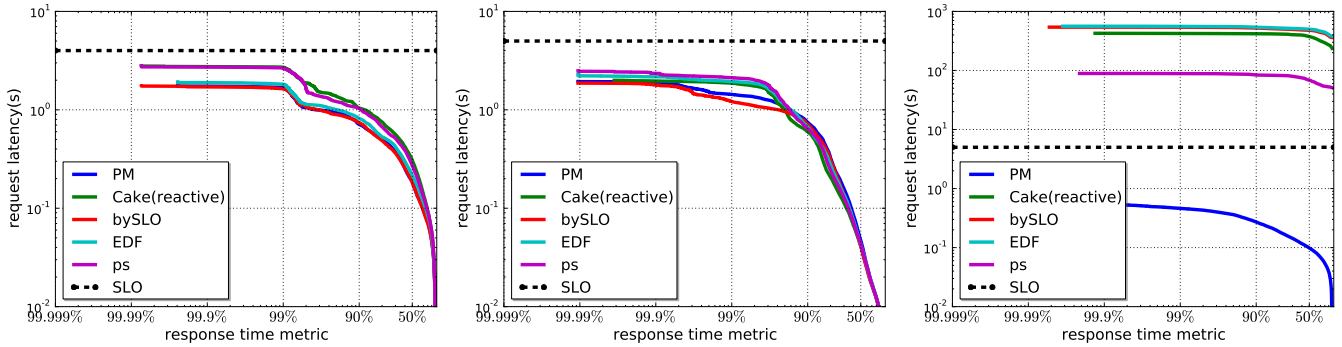


(a) low burstiness, $C_A^2 = 1$, Workload F (b) high burstiness, $C_A^2 = 20$, Workload G (c) very high burstiness, $C_A^2 = 40$, Workload H

Figure 7. Increased levels of burstiness affect both PriorityMeister (PM) and Cake, but PM meets the SLO at the 99th-%ile for inter-arrival burstiness levels up to $C_A^2 = 40$.

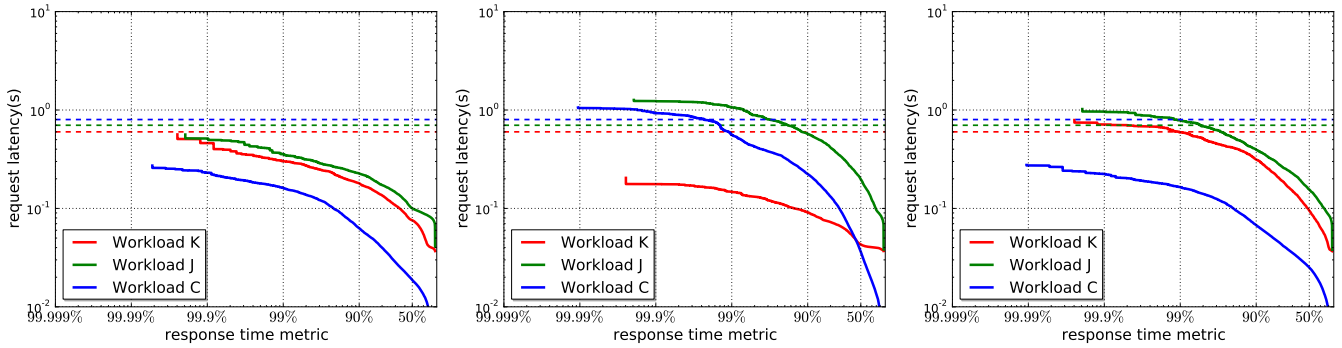
Fig. 6 shows a more descriptive representation of the latency (y-axis) at different percentiles (x-axis). It is essentially a representation of the CDF in log scale to focus on the tail behavior, with higher percentiles on the left. The results are grouped by scheduling policy, and it is easy to see that PriorityMeister is the only policy that doesn't violate any SLOs.

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 [21] 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 colocating 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



(a) Workload D, behaved (b) Workload C, behaved (c) Workload C when Workload D misbehaves

Figure 8. PriorityMeister is the only policy that isolates the effect of the misbehaving Workload D on Workload C.



(a) PM: all workloads' SLOs satisfied (b) bySLO: green SLO violated @ 95th-%ile (c) no QoS: green SLO violated @ 98th-%ile

Figure 9. When workloads induce a bottleneck on server network egress, PM is the only policy that meets all SLOs across all latency %iles.

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.

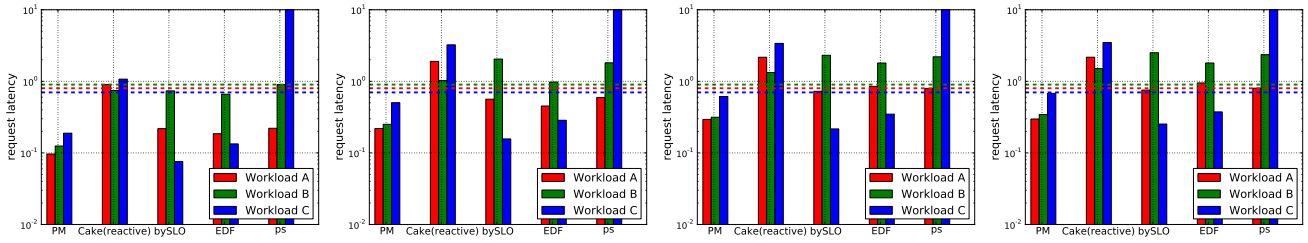
6.2 Coping with burstiness

In Fig. 7, 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 [21]. To vary the burstiness of a 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 continues to do worse, even dropping below 96%. PriorityMeister is better able to cope with the burstiness by prioritizing the latency sensitive workload, though there are cases (e.g., Fig. 7(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.

6.3 Misbehaving workloads

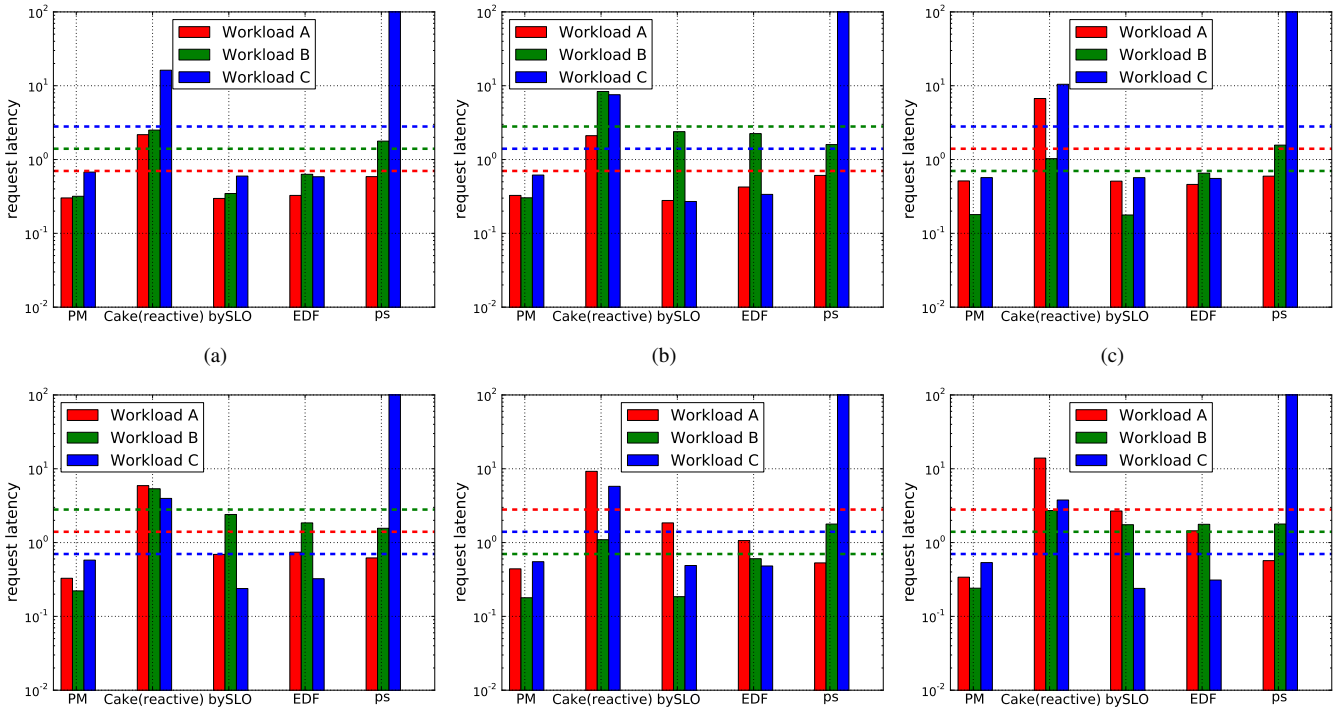
Since PriorityMeister is automatically configuring workload priorities, a natural question to ask is whether prioritization is safe. If a workload misbehaves and hogs the bandwidth, a good QoS system is able to contain the effect and avoid starving the other well-behaved workloads. PriorityMeister solves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

Fig. 8 demonstrates the effect of rate limiting with a two workload scenario where Workload D 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 (Fig. 8(a), Fig. 8(b)). However, when Workload D misbehaves and floods the system with requests, Workload C (Fig. 8(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) 90th-%ile (b) 99th-%ile (c) 99.9th-%ile (d) 99.99th-%ile

Figure 10. PriorityMeister is robust to storage latency mis-estimation (see Sec. 6.5). Same experiment as Fig. 5, but with a less accurate storage estimator. Despite the mis-estimation, results are similar.



(a) (b) (c) (d) (e) (f)

Figure 11. The 6 graphs each show a different permutation of 3 SLO values. 99.9th-%ile 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.

6.4 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 in these experiments 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. Fig. 9 shows an experiment with a server egress bottleneck where all the workloads are accessing storage devices colocated on a single machine. Client ingress bottleneck experiment results are similar, but are not shown due to space constraints. Both illustrate the need for network traffic conditioning. Without network QoS (Fig. 9(c)), workloads start missing their SLOs at the tail. PriorityMeister (Fig. 9(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 (Fig. 9(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 for timestamping packets and ordering packets by timestamp, which is not supported in network switches. Cake would require a mechanism for proportionally sharing the network, which is difficult to do in a distributed environment.

6.5 Sensitivity analysis

Estimator inaccuracy. Storage estimation is known to be a challenging problem, and we are interested in how well PriorityMeister performs when the estimates are inaccurate. To demonstrate the robustness to estimator inaccuracy, we replaced our default storage estimator with a simple estimator that assumes a constant base time for servicing a request. Fig. 10 shows the same experiment as in Sec. 6.1, but with the less accurate estimator. We find that PriorityMeister ends up selecting the same priority order and producing similar latency results.

While having more accurate storage estimates is better, PriorityMeister is not reliant upon having accurate storage estimates. PriorityMeister primarily uses estimates in determining priority orderings. As long as the same priority ordering is chosen, we expect similar latency performance.

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 Sec. 6.1 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. Fig. 11 shows the 99.9% latencies for these experiments. Surprisingly, we find that bySLO does a reasonable job at meeting SLOs for 5 of the 6 experiments. We believe that bySLO is a simple way of getting started and can act as a stand-in for our prioritizer algorithm. However, there are still some cases, as seen in this last example and earlier examples, where prioritizing by SLO does not work, and PriorityMeister is able to detect and correct for this.

7. Conclusion

PriorityMeister combines priorities and rate limits, automatically configured, to provide superior tail latency QoS for shared networked storage in cloud computing infrastructures. Previous approaches can work for non-bursty workloads, but struggle to cope with the burstiness that characterizes most real environments. Experiments with real workload traces on a real system show that PriorityMeister can provide even extreme tail latency SLOs, whereas the state-of-the-art previous approaches cannot.

Acknowledgments

We thank the member companies of the PDL Consortium (Actifio, APC, EMC, Facebook, Fusion-IO, Google, HP, Hitachi, Huawei, Intel, Microsoft, NEC Labs, NetApp, Oracle, Samsung, Seagate, Symantec, Western Digital) for their interest, insights, feedback, and support. This research is supported in part by Intel as part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC), by an NSERC Postgraduate Fellowship, and by the National Science Foundation under awards CSR-1116282, 0946825.

References

[1] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. 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.

[2] E. Anderson. Simple table-based modeling of storage devices, 2001.

[3] A. Bouillard, L. Jouhet, and E. Thierry. Tight performance bounds in the worst-case analysis of feed-forward networks. In *Proceedings of the 29th Conference on Information Communications*, INFOCOM'10, pages 1316–1324, Piscataway, NJ, USA, 2010. IEEE Press. ISBN 978-1-4244-5836-3. URL http://dl.acm.org/citation.cfm?id=1833515.1833708.

[4] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013. ISSN 0001-0782. . URL http://doi.acm.org/10.1145/2408776.2408794.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. . URL http://doi.acm.org/10.1145/1294261.1294281.

[6] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. 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.

[7] A. Ghodsi, V. Sekar, M. Zaharia, and I. 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. . URL http://doi.acm.org/10.1145/2342356.2342358.

[8] A. Gulati, A. Merchant, and P. J. Varman. pclock: an arrival curve based approach for qos guarantees in shared storage systems. In *Proceedings of the 2007 ACM SIGMETRICS in-*

ternational 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. . URL http://doi.acm.org/10.1145/1254882.1254885.

[9] A. Gulati, I. Ahmad, and C. 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.

[10] A. Gulati, A. Merchant, and P. 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.

[11] W. Jin, J. S. Chase, and J. 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. . URL http://doi.acm.org/10.1145/1005686.1005694.

[12] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance differentiation for storage systems using adaptive control. *Trans. Storage*, 1(4):457–480, Nov. 2005. ISSN 1553-3077. . URL http://doi.acm.org/10.1145/1111609.1111612.

[13] S. Kavalanekar, B. L. Worthington, Q. Zhang, and V. Sharda. Characterization of storage workload traces from production windows servers. In D. Christie, A. Lee, O. Mutlu, and B. G. Zorn, editors, *IISWC*, pages 119–128. IEEE, 2008. ISBN 978-1-4244-2778-9. URL http://dblp.uni-trier.de/db/conf/iiswc/iiswc2008.html#KavalanekarWZS08.

[14] J.-Y. Le Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, Berlin, Heidelberg, 2001. ISBN 3-540-42184-X.

[15] C. R. Lumb, A. Merchant, and G. 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.

[16] A. Merchant, M. Uysal, P. Padala, X. Zhu, S. Singhal, and K. 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. . URL http://doi.acm.org/10.1145/1998582.1998638.

[17] B. Schroeder, A. Wierman, and M. 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.

[18] D. Shue, M. J. Freedman, and A. 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.

[19] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. 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. . URL http://doi.acm.org/10.1145/2517349.2522723.

[20] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. 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.

[21] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. 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. . URL http://doi.acm.org/10.1145/2391229.2391243.

[22] J. Wu and S. 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.

[23] Y. Xu, M. Bailey, B. Noble, and F. 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. . URL http://doi.acm.org/10.1145/2523616.2523620.

[24] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 329–342, Berkeley, CA, USA, 2013. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2482626.2482658.

[25] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage performance virtualization via throughput and latency control. *Trans. Storage*, 2(3):283–308, Aug. 2006. ISSN 1553-3077. . URL http://doi.acm.org/10.1145/1168910.1168913.