WorkloadCompactor: Reducing datacenter cost while providing tail latency SLO guarantees

Timothy Zhu
The Pennsylvania State University

Michael A. Kozuch
Intel Labs

Mor Harchol-Balter
Carnegie Mellon University

ABSTRACT

Service providers want to reduce datacenter costs by consolidating workloads onto fewer servers. At the same time, customers have performance goals, such as meeting tail latency Service Level Objectives (SLOs). Consolidating workloads while meeting tail latency goals is challenging, especially since workloads in production environments are often bursty. To limit the congestion when consolidating workloads, customers and service providers often agree upon rate limits. Ideally, rate limits are chosen to maximize the number of workloads that can be co-located while meeting each workload's SLO. In reality, neither the service provider nor customer knows how to choose rate limits. Customers end up selecting rate limits on their own in some ad hoc fashion, and service providers are left to optimize given the chosen rate limits.

This paper describes WorkloadCompactor, a new system that uses workload traces to automatically choose rate limits simultaneously with selecting onto which server to place workloads. Our system meets customer tail latency SLOs while minimizing datacenter resource costs. Our experiments show that by optimizing the choice of rate limits, WorkloadCompactor reduces the number of required servers by 30-60% as compared to state-of-the-art approaches.

CCS CONCEPTS

• **Information systems** → **Network attached storage**; • **Computer systems organization** → *Cloud computing*; • **Software and its engineering** → *Software system models*;

KEYWORDS

workload consolidation, tail latency, quality of service, networked storage, network calculus, linear programming

ACM Reference Format:
Timothy Zhu, Michael A. Kozuch, and Mor Harchol-Balter. 2017. WorkloadCompactor: Reducing datacenter cost while providing tail latency SLO guarantees. In *Proceedings of ACM Symposium of Cloud Computing conference, Santa Clara, California USA, September 25-27 (SoCC'17),* 13 pages. https://doi.org/10.1145/3127479.3132245

1 INTRODUCTION

In cloud computing and enterprise datacenter environments, service providers often seek to maximize the utilization of their resources by *sharing* compute, network, and storage resources among customers. At the same time, service providers want to keep their customers happy by providing good performance. Some customers may specify their performance goals in terms of a *tail latency* Service Level Objective (SLO), such as "99% of requests must complete within 150 milliseconds". Cloud researchers and companies such as Amazon and Google have repeatedly stressed the importance of meeting tail latency SLOs at, for example, the 99th and 99.9th percentiles, particularly for user-facing interactive applications [2, 5, 6, 9, 13, 17, 20, 21, 25, 26].

Our work is designed for long-running user-facing applications, such as web servers and email servers, in the context of network attached storage, such as Amazon's Elastic Block Store (EBS). We address the problem of how to consolidate multiple workloads onto a storage server while meeting tail latency performance goals. Specifically, each workload sends a stream of requests to a storage server containing its data. Our goal is to (1), ensure that each workload's stream of requests meets its tail latency SLO, while (2), minimizing the number of servers that the service provider uses to satisfy all the given workloads.

Our work addresses two key questions: what portion of a server (e.g., storage throughput, network bandwidth) should be allocated to each storage workload, and on which server should its data be placed. The current practice for allocating storage resources is to have customers either reserve some amount of storage throughput (e.g., Amazon's Provisioned IOPS) or run without any guarantees in a best effort fashion. Today's cloud providers do not support tail latency SLOs since they are much harder to guarantee than throughput, especially with bursty workloads. Unfortunately, using throughput reservations to meet tail latency SLOs is inefficient because the bursty nature of a workload's traffic induces customers to reserve more throughput than necessary.

A key challenge we address is managing the short-term burstiness[1] that is commonly exhibited by production workloads. Even when a server is not overloaded, short-term burstiness can have a significant effect on tail-latency [25], and thus the ability to co-locate workloads while meeting tail latency SLOs. In this work, we focus on short-term burstiness and show how to characterize short-term burstiness and quantify its effect on tail latency.

To handle bursts while guaranteeing SLOs, multiple research papers [9, 11, 13, 24] have proposed using token bucket rate limiting (Fig. 1) with both a rate (i.e., throughput) parameter (r) and a burst (a.k.a. bucket size) parameter (b). Note that there are two

[1] Burstiness occurs at different time granularities. Short-term burstiness occurs at the granularity of seconds or milliseconds, diurnal patterns occur at the granularity of a day, and long-term load variations occur at the granularity of months.

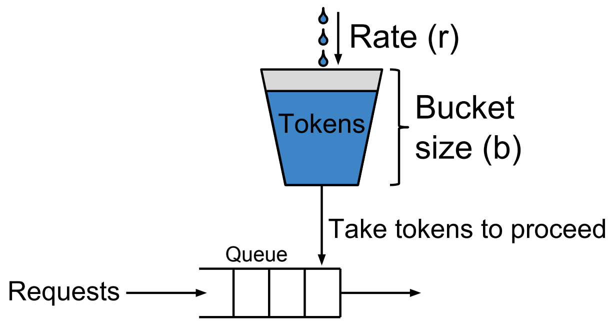


Figure 1: Token bucket rate limiters control the rate and burstiness of a stream of requests. When a request arrives at the rate limiter, tokens are used (i.e., removed) from the token bucket to allow the request to proceed. If the bucket is empty, the request must queue and wait until there are enough tokens. Tokens are added to the bucket at a constant rate *r* up to a maximum capacity as specified by the bucket size *b*. Thus, the token bucket rate limiter limits the workload to a maximum instantaneous burst of size *b* and an average rate *r*.

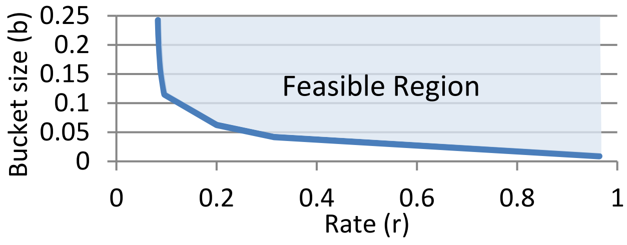


Figure 2: Example *r*-*b* tradeoff curve.

perspectives on rate limits. From the perspective of the customer, the rate limit describes the traffic capacity reserved for its workload, *w*. From the perspective of the provider, the rate limit serves as an upper bound on *w*'s traffic demand that the provider must support – and hence, the rate limit on *w* ensures that sufficient resource bandwidth remains to support the guarantees provided to the other workloads co-located with *w*. Rate limiting has been successfully used to provide network latency guarantees in Silo [13] and QJump [9] and storage latency guarantees in pClock [11] and Avatar [24]. However, an open problem in these papers is a method for how to *choose* the rate limit parameters. These systems assume the customer provides the rate limit parameters as input.

Impact of rate limits on consolidation

The key insight in our work is jointly optimizing the choice of $\langle r, b\rangle$ rate limit parameters for each workload to better compact workloads onto servers. Our results show that selecting rate limit parameters is an important problem that can result in using many more servers than necessary. The reason for such a big difference is because there are (infinitely) many feasible $\langle r, b\rangle$ choices for a given workload, and the choice of an $\langle r, b\rangle$ tuple affects how easily it can be co-located with other workloads. We define *feasible* $\langle r, b\rangle$ tuples

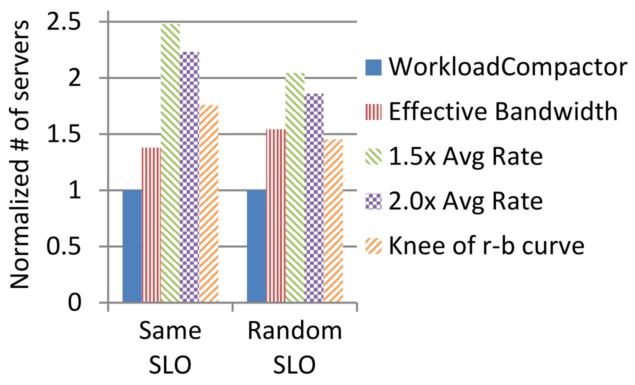


Figure 3: Comparing WorkloadCompactor to state-of-the-art approaches under two scenarios, each using 1000 workloads based on production traces. In the first scenario, all workloads specify the same SLO, and in the second scenario, workloads specify random SLOs. Results are normalized to the number of servers used by WorkloadCompactor to clearly show that state-of-the-art approaches require 40-150% more servers than WorkloadCompactor.

for a workload as rate limit parameters that are high enough such that the rate limiter does not delay workload requests. Fig. 2 shows an example of feasible $\langle r, b\rangle$ tuples where all points on or above the *r*-*b* curve are feasible. *We use the r-b curve as a characterization of workload burstiness.*

The *r*-*b* curve represents a tradeoff in characterizing burstiness via the rate *r* parameter and bucket size *b* parameter. The *r*-*b* curve notion was introduced in [26] and is an alternative representation of the arrival curve concept in network calculus theory. Formally, a workload with an $\langle r, b\rangle$ rate limit is by definition constrained by the token bucket to never send more than

$$r \cdot t + b$$

units of work within all time windows of length *t*, for all *t*. From this equation, a workload's traffic burstiness can either be accounted for by adjusting its *b* parameter, which is good for instantaneous bursts, or by adjusting its *r* parameter, and the *r*-*b* curve quantifies this tradeoff between *r* and *b*.

Having the *r*-*b* curve rather than a single $\langle r, b\rangle$ tuple is important since being able to choose each workload's $\langle r, b\rangle$ tuple has a significant impact on the ability to co-locate workloads. For example, if all workloads select rate limits with low rates and large bucket sizes, then it will be hard to co-locate workloads since the large bucket sizes will allow large bursts that could cause SLO violations. Similarly, if all workloads select rate limits with high rates and small bucket sizes, then it will be hard to co-locate workloads since the available bandwidth will quickly be used up.

In Fig. 3, we compare multiple approaches to choosing an $\langle r, b\rangle$ tuple. One natural approach is to select rate limits by setting *r* to the average rate multiplied by some constant *k* (e.g., $k = 1.5$). The bucket size *b* can then be set by trial and error experiments or via the *r*-*b* curve. This approach has been used, for example, by the authors of the recent Silo [13] paper in their experiments. We refer to this approach as 1.5x Avg Rate or 2.0x Avg Rate, corresponding to the values used in Silo. Our test results with a range of values from $k = 1.25$ to $k = 20$ are not significantly better. Another natural heuristic, which we call "Knee of *r-b* curve", is to select the knee of the *r-b* curve in hopes of making a good tradeoff between r and b. The state-of-the-art in theory for selecting rate limits is based on the Effective Bandwidth approach from network calculus [15], which has been shown to be reasonable in [25]. Unfortunately, all of these approaches are quite suboptimal in minimizing the number of servers. This is because there is no "best" $\langle r, b \rangle$ tuple for a workload; the optimal $\langle r, b \rangle$ tuple depends on the other workloads sharing the server.

WorkloadCompactor

WorkloadCompactor is a new system that automatically selects $\langle r, b \rangle$ rate limits for each workload in conjunction with deciding onto which server to place workloads to meet tail latency SLOs. WorkloadCompactor first associates an *r-b* curve with each workload as a characterization of the workload's behavior. The *r-b* curve for a workload is generated via WorkloadCompactor's `rbGen` tool, which takes as input a historic trace of that workload's behavior. We assume the trace is long enough to represent an upper bound on the workload's behavior. Clearly, it is impossible to ensure tail latency SLOs without an upper bound on the expected traffic. For example, if the workload exhibits diurnal variations, then the trace should cover a typical full day of time or a high load period of the day. The customer can also add a "safety margin" to the workload's *r-b* curve to account for load variations over longer time periods (see Sec. 5.2).

What makes WorkloadCompactor unique is that in picking rate limits, it *simultaneously* considers the *r-b* curves for *all* workloads sharing a server. This is in contrast to prior approaches which select a rate limit independently for each workload, not taking the other workloads into account. Specifically, when WorkloadCompactor places a new workload onto a server, it *dynamically reconfigures* the $\langle r, b \rangle$ rate limit for *each* workload on that server so that the new workload can fit; the newly reconfigured rate limits are chosen from each workload's *r-b* curve. The key insight that makes the dynamic rate limit reconfiguration fast is that we can represent the joint optimization problem as a *specially formed linear program (LP) based on equations from network calculus.* WorkloadCompactor also provides a scalable heuristic for quickly deciding onto which server to place workloads.

This paper makes the following main contributions:

- **Building an automated system for minimizing the number of servers to meet tail latency SLOs:**
  WorkloadCompactor is a new QoS system that enforces rate limits and priorities in storage and network to meet tail latency SLOs. WorkloadCompactor minimizes the number of servers using a new technique for automatically selecting rate limits and priorities to compact more workloads onto a server while meeting SLOs; our technique is based on non-trivial applications of network calculus. Our compaction technique is used in conjunction with our scalable placement algorithm, which places workloads onto servers an order of magnitude faster than the traditional first-fit policy.
- **Extensive evaluation:**
  We evaluate WorkloadCompactor on a physical 24-machine cluster using 62-85 workloads derived from real production traces to demonstrate that WorkloadCompactor uses 30-60% fewer servers than state-of-the-art approaches while meeting tail latency SLOs. Our scalability experiments with 1000 workloads show that WorkloadCompactor is able to quickly and effectively pack workloads at large scale. We also show that WorkloadCompactor works well in a broad range of scenarios such as mixing workload arrivals/departures and using multiple SSDs per server.
- **Open-source implementation:**
  Code for WorkloadCompactor is available for public use at https://github.com/timmyzhu/WorkloadCompactor. WorkloadCompactor is designed to integrate into existing storage and/or network infrastructures (e.g., Amazon EBS, OpenStack Cinder, IOFlow [19]) as an add-on feature for controlling traffic to meet tail latency SLOs. In our implementation, we integrate WorkloadCompactor with NFS and Linux TC to enforce rate limits and priorities at storage and network devices.

2 ARCHITECTURE

2.1 System overview

In this work, we target storage workloads, which send a stream of requests (e.g., read, write) over the network to access data on storage servers. We imagine Amazon's Elastic Block Store (EBS) as a typical example scenario for WorkloadCompactor, but the techniques described are applicable to other systems such as NFS servers, memcached servers, or databases. In this example scenario, an Amazon customer runs a workload (e.g., mail server) on an "instance"[2] connected to one or more "EBS volumes". An EBS volume is hosted on a storage server, which provides networked storage to the volume's connected instance. WorkloadCompactor is responsible for helping the service provider, Amazon in this case, decide onto which storage server to host an EBS volume along with rate limits for workloads accessing the storage server.

Fig. 4 shows the process of adding a workload in WorkloadCompactor. When a customer wishes to add a workload (e.g., a database backed by an EBS volume), the customer allocates an instance for the database in the usual way. However, when the customer allocates the EBS volume, the customer specifies the desired latency along with a description of the workload's storage and network utilization in the form of *r-b* curves. The customer generates the *r-b* curves via our provided `rbGen` tool (Sec. 2.2) using historic traces. Note that if a workload's behavior changes significantly, the other workloads sharing the system are still protected, because rate limits are individually enforced for each workload. If the workload's new behavior is expected to continue, then the customer would need to regenerate an *r-b* curve based on the new behavior and submit it to WorkloadCompactor, which may trigger a migration of the workload. The customer can also scale the *r-b* curves to include a safety margin for deviations in past behavior. We are motivated by workloads, such as Wikipedia, where daily peak request rates

[2]Instances represent virtual machines (VMs) in Amazon, but the design of WorkloadCompactor does not require virtualization.

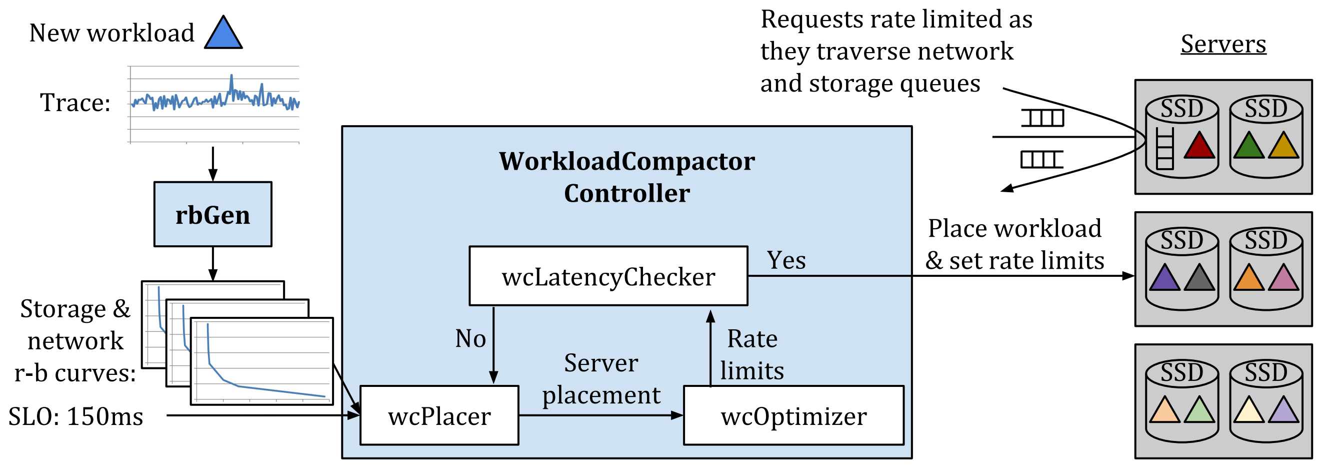

New workload
Trace:
rbGen
Storage & network r-b curves:
SLO: 150ms
WorkloadCompactor Controller
wcLatencyChecker
Yes
No
Rate limits
Server placement
wcPlacer
wcOptimizer
Requests rate limited as they traverse network and storage queues
Servers
SSD
Place workload & set rate limits

Figure 4: WorkloadCompactor system diagram.

are stable across months (see Sec. 5.2 and Fig. 9); we explore the robustness of *r*-*b* curves in Sec. 5.2. The provider then provides the desired level of service as specified by the SLO and *r*-*b* curves.

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

2.2 *rbGen*: Generating *r*-*b* curves

To simplify the description of the `rbGen` algorithm, we present an alternate, but equivalent, description of token bucket rate limiting than the traditional description in Fig. 1. In the alternate description, when a request arrives, tokens are *added* to the token bucket. If there is sufficient space to add tokens to the token bucket without exceeding the bucket size *b*, then the request is allowed to proceed. Otherwise, the request is queued and waits until there is sufficient space. Space becomes available as tokens continuously *drain* from the bucket at the configured rate *r*.

Algorithm 1 provides the pseudocode for `rbGen`, WorkloadCompactor's tool for generating *r*-*b* curves based on a given trace. Traces contain a list of requests parameterized by the arrival time, request type (e.g., read, write), and request size (e.g., 4KB). The tool sweeps across a given list of *r* values (e.g., 0.1, 0.2, ..., 1.0), and for each *r* value, it computes the minimum *b* such that the requests are not queued. These *b* values are computed by replaying the trace with an infinite sized token bucket (`virtualBucket`) at each rate *r* and

Algorithm 1: ***r*-*b* curve generation**

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

tracking the maximum tokens in the bucket at any point in time. The output *b* values along with the input *r* values then form the $\langle r, b\rangle$ vertices in the piecewise linear *r*-*b* curve. To simplify the mathematics, all *r*-*b* curves are normalized (e.g., divide by network link bandwidth) such that $r = 1.0$ represents a workload requesting 100% utilization.

Note that different *r*-*b* curves need to be generated for network traffic into the server, storage traffic at the server, and network traffic out of the server. Network traffic into and out of the server are accounted for separately since the amount of data transferred depends on request type (e.g., read/write). `rbGen` generates different *r-b* curves via the `tokensFunc` parameter, which converts request sizes to tokens. For example, the network traffic leaving the server would use the number of bytes accessed for read requests and a constant (i.e., size of acknowledgment) for write requests. For storage, we implement a storage model (Sec. 3.1) to represent the amount of "work" introduced by a request. WorkloadCompactor is designed to handle the complexity of multiple stages of traffic, and it automatically selects the $\langle r, b\rangle$ rate limits for each stage while accounting for the end-to-end latency across stages.

2.3 *wcLatencyChecker*: Guaranteeing SLOs

WorkloadCompactor relies upon network calculus, which has been shown to be effective in related literature [9, 13, 26]. Network calculus provides a framework for calculating latency guarantees based on the selected $\langle r, b\rangle$ tuples. Specifically, we use network calculus equations to compute the latency due to queueing at a server; we write equations from the perspective of a single server and repeatedly apply them to each server.

To simplify exposition, we first show how to handle a single stage (e.g., storage) and later show how to extend the analysis to multiple stages. For *any* workload at priority p, an upper bound on the workload's tail latency is:

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

where $\langle r_j, b_j\rangle$ corresponds to workload j's selected rate limit. Equation (1) is derived from network calculus basic principles [15]. The numerator is the sum of bucket sizes b_j across workloads j where j's priority, denoted by p_j, is higher than or equal to p. The denominator is 1 minus the sum of rates r_j across workloads j where j's priority p_j is strictly higher than p.

From Equation (1), note that prioritization provides the benefit that workloads are only affected by equal or higher priority workloads. WorkloadCompactor uses prioritization to provide better latency for the workloads with tighter SLO constraints. Specifically, WorkloadCompactor sets priorities in order of SLOs such that workloads with tighter SLOs are assigned higher priorities, as proposed in [26]. In other words, each priority p is associated with a SLO, denoted by SLO_p, where $p1 > p2$ implies $SLO_{p1} < SLO_{p2}$.

2.3.1 wcOptimizer*: Selecting optimal rate limits.*

The choice of $\langle r, b\rangle$ parameters has a significant impact on how many workloads can be co-located onto servers. Rather than using ad hoc approaches to choose the rate limit parameters, WorkloadCompactor introduces a novel systematic approach for optimizing the $\langle r, b\rangle$ parameters; existing strategies are described in Sec. 4.1. Our approach is based on two key ideas.

First, since WorkloadCompactor associates an *r-b* curve with each workload, when a new workload is added to a server, WorkloadCompactor is able to dynamically recompute rate limits for all existing workloads sharing that server. Thus, WorkloadCompactor does not need to consider future workload arrivals and only needs to optimize based on the current workloads in the system.

Second, WorkloadCompactor directly embeds Equation (1) into its optimization. Since Equation (1) is used to check if workloads can be co-located, WorkloadCompactor can check if there exists any set of rate limit parameters for the workloads such that they all can be co-located. While checking all possible rate limits may sound slow and intractable, a key insight is that we can actually represent the problem as a linear program (LP), which can be efficiently solved. Specifically, for each priority level p with a given SLO, SLO_p, we want to ensure that:

$$\frac{\sum_{j|p_j \geq p} b_j}{1 - \sum_{j|p_j > p} r_j} \leq SLO_p \tag{2}$$

which can be rewritten as the linear inequality:

$$\sum_{j|p_j \geq p} b_j + \sum_{j|p_j > p} r_j \cdot SLO_p \leq SLO_p \tag{3}$$

Thus, WorkloadCompactor creates an LP with r_j and b_j as LP variables representing workload j's selected rate limit $\langle r_j, b_j\rangle$. Equation (3) is added as a constraint for each priority level to ensure SLOs are guaranteed. Additionally, constraints are added to ensure that each selected rate limit $\langle r_j, b_j\rangle$ is on (or above) the workload's *r-b* curve. Since the *r-b* curves are piecewise linear convex functions, they can be encoded as linear constraints in the LP by taking each of the lines defined by the piecewise segments in the *r-b* curve and adding an LP constraint that $\langle r_j, b_j\rangle$ is above the line. Lastly, the following LP constraint is added to ensure the server is not overloaded:

$$\sum_j r_j \leq 1 \tag{4}$$

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

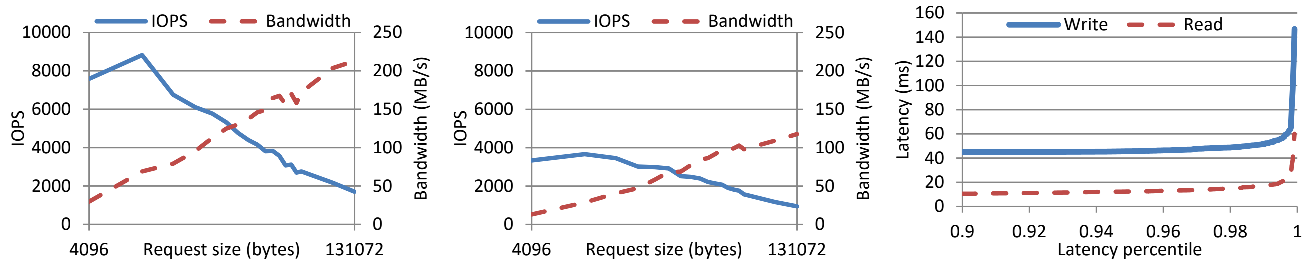
WorkloadCompactor then uses an off-the-shelf solver (e.g., GLPK) to determine if the LP is feasible (i.e., there exist valid $\langle r_j, b_j\rangle$ rate limits that satisfy the constraints) or if there are no such rate limit configurations that can satisfy all workload SLOs. Since LP feasibility is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

To handle multiple stages (e.g., network, storage), WorkloadCompactor uses Equation (1) three times to represent the three stages: network into server, storage, network out of server. This results in the following equation:

$$\frac{\sum_{j|p_j \geq p} b_j^{netIn}}{1 - \sum_{j|p_j > p} r_j^{netIn}} + \frac{\sum_{j|p_j \geq p} b_j^{storage}}{1 - \sum_{j|p_j > p} r_j^{storage}} + \frac{\sum_{j|p_j \geq p} b_j^{netOut}}{1 - \sum_{j|p_j > p} r_j^{netOut}} \leq SLO_p \tag{5}$$

Unfortunately, Equation (5) is not a linear inequality, which makes the optimization difficult. The key trick we discovered in solving this problem is to apply a relaxation to the problem to convert it into a linear inequality. Specifically, we add a new LP variable R_p for each priority level such that it obeys the following three constraints:

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



(a) Read throughput (IOPS/Bandwidth) (b) Write throughput (IOPS/Bandwidth) (c) Tail latency

Figure 5: Performance profile of NFS storage stack running with our SSD.

Intuitively, the R_p variable balances the rate across the three stages. Equation (5) can then be relaxed to the inequality:

$$\frac{\sum_{j|p_j \geq p} b_j^{netIn}}{1 - R_p} + \frac{\sum_{j|p_j \geq p} b_j^{storage}}{1 - R_p} + \frac{\sum_{j|p_j \geq p} b_j^{netOut}}{1 - R_p} \leq SLO_p \qquad (6)$$

which can be rewritten as the linear inequality:

$$\sum_{j|p_j \geq p} b_j^{netIn} + \sum_{j|p_j \geq p} b_j^{storage} + \sum_{j|p_j \geq p} b_j^{netOut} + R_p \cdot SLO_p \leq SLO_p \qquad (7)$$

Thus, to handle multiple stages, WorkloadCompactor replaces Equation (3) with Equation (7), and for each priority level p, it adds the R_p variable along with R_p's 3 constraints.

2.4 *wcPlacer*: Selecting workload placements

Since storage workloads are difficult to migrate, we restrict our design space to solutions that do not rely upon constantly migrating workloads to fix bad placements. So to make a good placement where SLOs are met, WorkloadCompactor places workloads onto servers where they fit, as determined by solving the LP (Sec. 2.3.1). It remains to establish the order in which to check servers for fit. Our tests with placement heuristics[3] indicate that first-fit yields good packings, which agrees with theoretical results[4]; hence, WorkloadCompactor adopts a first-fit strategy.

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

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

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

3 PROTOTYPE IMPLEMENTATION

In implementing our WorkloadCompactor prototype, there are multiple challenges we face in working with storage tail latency. First, we describe how we model storage in WorkloadCompactor. Second, we describe how we enforce priorities in WorkloadCompactor.

3.1 Storage model

Though our techniques can be extended to various storage devices, our WorkloadCompactor prototype focuses on Solid-State Drives (SSDs). In this section, we describe how we model the performance of SSDs and the storage stack. SSDs are complex devices with many performance peculiarities, making them difficult to model. SSD performance cannot be described with a single parameter, but rather requires profiling the device across various access types. For example, read and write throughput is very different for SSDs. Writes may need to erase SSD blocks, which is considerably slower than reading SSD blocks. To accurately profile a SSD, we profile reads and writes separately. Additionally, the request size significantly impacts SSD throughput. For small requests, SSDs are limited to the maximum IOPS supported by the device. For large requests, SSDs are limited to the maximum bandwidth supported by the device. WorkloadCompactor builds a performance profile (e.g., Fig. 5(a) and Fig. 5(b)) for each SSD by measuring the empirical throughput over a sweep of request sizes. The performance profile includes the performance of both the SSD and storage stack so as to have a holistic view of the storage subsystem. We assume the storage performance profile does not significantly change over time, and variation in performance can be addressed by using more conservative (i.e., lower) throughput numbers.

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

In addition to the work generated by a request, there is also a tail latency effect due to the SSD and storage stack. For example, writes are sometimes delayed to allow more write batching, and SSD garbage collection is known to affect tail latency. Thus, WorkloadCompactor also profiles the tail latency of requests without the queueing effects of bursty workloads to isolate the SSD and storage stack tail latency (e.g., Fig. 5(c)). This profiled latency is then added to the estimated queueing latency from the network calculus equations. WorkloadCompactor focuses on limiting the tail latency due to queueing and is complementary to other works that focus on device tail latencies (e.g., [22]).

3.2 Enforcing priorities in SSDs

To enforce priorities (and rate limits) in storage, we implement a prototype storage enforcement module. The most straightforward way to enforce priorities at the SSD is to dispatch one request at a time to the SSD. However, dispatching requests one at a time does not work well for SSDs because modern SSDs require a high degree of parallelism to achieve high throughput[5].

While parallelizing requests enables high throughput for SSDs, it also has the potential to interfere with the priority ordering of WorkloadCompactor. When a high priority request arrives at the storage system, it may need to wait for outstanding low priority requests. Also, SSDs may unintentionally delay a high priority request in order to more efficiently serve low priority requests. This can induce starvation for high priority requests while other requests are being served [23].

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

4 EXPERIMENTAL SETUP

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

4.1 Comparison approaches

To evaluate the effectiveness of WorkloadCompactor, we compare its performance to three state-of-the-art approaches to selecting a workload's rate limits: scaling average bandwidth, effective bandwidth, and finding the knee of the r-b curve. To make a fair comparison, all approaches provide tail latency SLO guarantees by adhering to Equation (1). Workloads are placed using a first-fit strategy, which works well, as noted in Sec. 2.4.

4.1.1 Scaling average bandwidth. Little is known about selecting rate limits, and most users resort to ad hoc heuristics. Authors of the recent Silo [13] paper, for example, select rate limits by setting r to the average rate of the workload multiplied by some constant k (e.g., $k = 1.5$). The b parameter can then be determined through trial and error experiments or via the r-b curve (Sec. 2.2). By choosing higher r values, smaller bursts are allowed into the system, which allows more workloads to be co-located without violating SLOs. However, higher r values may also exhaust the available bandwidth. Our results evaluate this approach with two values of k: 1.5 and 2, corresponding to values used in Silo. We also test a range of values from 1.25 to 20, but find that all of them perform worse than the effective bandwidth approach, described next.

4.1.2 Effective bandwidth. The state-of-the-art in selecting rate limits is based on the effective bandwidth theory [15] and has shown to be a reasonable approach in [25]. The effective bandwidth approach is designed to isolate each workload's burstiness from the other workloads in the system. Intuitively, the effective bandwidth approach slows down traffic at the rate limiter to create smooth traffic and eliminate burstiness within the system. Thus, the effective bandwidth approach sets b to 0 to create smooth traffic and calculates the minimum r (known as the effective bandwidth) such that the workload is slowed down by no more than the SLO.

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

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

4.2 Traces

Our evaluation uses a collection of real production storage traces of Microsoft services (e.g., LiveMaps, Exchange), which are described in detail in [14]. In our experiments, we consider each trace to represent a workload. Half of the trace is used for generating r-b curves (Sec. 2.2), and the other half is replayed on our cluster to demonstrate that WorkloadCompactor is able to meet tail latency SLOs. We replay traces in an open loop fashion, which properly captures the end-to-end latency and the effects of queueing.

[5]The reason behind this is that individual flash memory packages offer limited bandwidth which is commonly solved by bundling many packages together. In particular, modern SSDs employ parallelism at many levels (e.g., channel-level, package-level, die-level etc.) [4, 7].

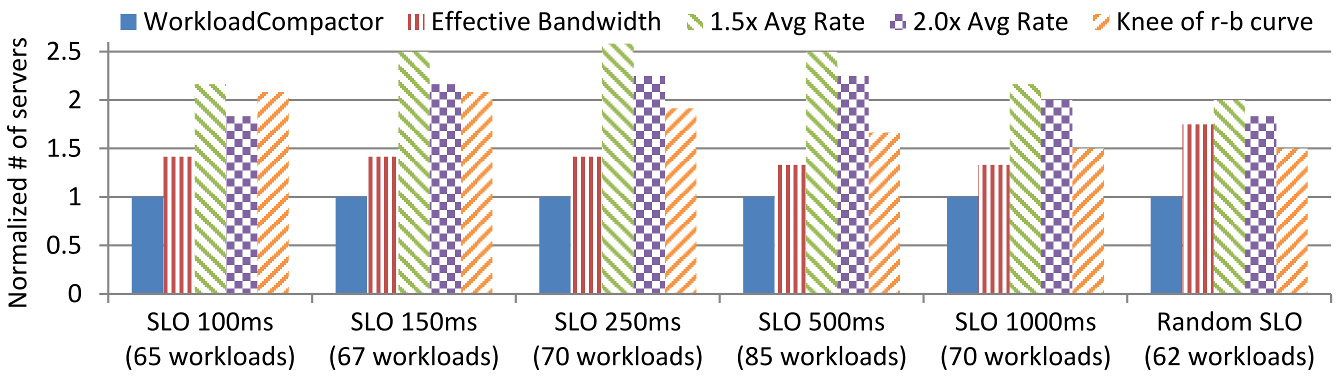


Figure 6: Number of servers required by state-of-the-art approaches to meet tail latency SLOs, normalized to the number of servers used by WorkloadCompactor (12 servers). In all experiments, we randomly select workloads. In the first 5 "Same SLO" experiments, we use a fixed SLO for all workloads. In the last "Random SLO" experiment, workloads are configured with random SLOs from {100ms, 150ms, 250ms, 500ms, 1000ms}. Each of these experiments is run on our local cluster, and WorkloadCompactor is able to meet all workload SLOs while using significantly fewer servers.

4.3 Experimental testbed

All experimental results are run on a dedicated 24-machine cluster, with 12 client machines and 12 server machines. Each machine is configured with two Intel Xeon E5-2680 processors, 64GB of DRAM, and an Intel 710 series 300GB SSD, and is connected via a 1Gbps network. Each machine runs 64-bit Ubuntu and provides virtualization support via the standard kvm package (qemu-kvm-1.0). We replay traces in VMs running 64-bit Ubuntu 14.04 and use the standard NFSv3 server and client that come with these operating systems to provide remote storage access.

5 RESULTS

5.1 WorkloadCompactor uses fewer servers

One of the surprising results in our work is that the ability to compact workloads onto servers while meeting tail latency SLOs is highly influenced by how rate limits are chosen for each workload. Fig. 6 compares WorkloadCompactor with the state-of-the-art approaches in choosing rate limits across several experiments. In each experiment, we assign 99.9% tail latency SLOs to randomly selected workloads and count the number of servers used, normalized to the number of servers used by WorkloadCompactor (12 servers). In the first 5 "Same SLO" experiments, we use a fixed SLO for all workloads. In the last "Random SLO" experiment, we assign random SLOs from {100ms, 150ms, 250ms, 500ms, 1000ms}. When selecting workloads, we only consider workloads that can meet their SLOs when run in isolation to avoid using a SLO that is too tight for a workload. As a result, in experiments with higher SLOs, we randomly select from a larger pool of workloads that includes more bursty workloads.

Fig. 6 shows that WorkloadCompactor uses far fewer servers than the state-of-the-art approaches. For the Same SLO experiments, effective bandwidth works better than the other state-of-the-art approaches, but still uses 40% more servers than WorkloadCompactor. For the Random SLO experiment, the knee method works better than effective bandwidth since the effective bandwidth approach is fundamentally unable to take advantage of prioritization benefits. Nevertheless, the knee method still uses 50% more servers than WorkloadCompactor. WorkloadCompactor is the only method that works well in all cases.

5.2 Robustness

To demonstrate that WorkloadCompactor meets 99.9% tail latency SLOs, we measure each workload's 99.9% latency when running the experiments in Sec. 5.1 on our local cluster. Our initial results (not shown) reveal that WorkloadCompactor meets all workload SLOs when workloads are represented by their *r-b* curves. To explore the effect when workloads deviate from their expected behavior, we run another set of experiments where we use the first half of each workload's trace to generate *r-b* curves and replay the second half. We find that almost all workloads still meet their SLOs, but a few miss their SLOs due to specifying *r-b* curves that are too small. One way of addressing this issue is to add a "safety margin" by increasing the *r-b* curves. Fig. 7 and Fig. 8 show our experimental results with a 10% safety margin (i.e., scaling the *r-b* curves by 1.1); all of the workload 99.9% latencies (vertical bars) are under the SLO (horizontal line) in all experiments.

The selection of a "safety margin" is left to the customer, and it depends on how likely the workload's behavior is expected to change over the long-term and the customer's risk adversity. Our work is motivated by the fact that production workloads are often very bursty at short time scales (as seen in all of our production workload traces), but are more stable over longer time periods. As a motivating example of long-term stability, Fig. 9 shows a timeline graph of Wikipedia english traffic in 2017 [1]. The graph shows that each day's maximum hourly request rate is stable over a span of multiple months. The highest request rate is within 15% of the median day's maximum hourly request rate. Importantly, while many workloads exhibit long-term stability, there is significant short-term burstiness in the traces we've analyzed, which has a large effect on tail latency and the ability to pack workloads while satisfying tail latency SLOs.

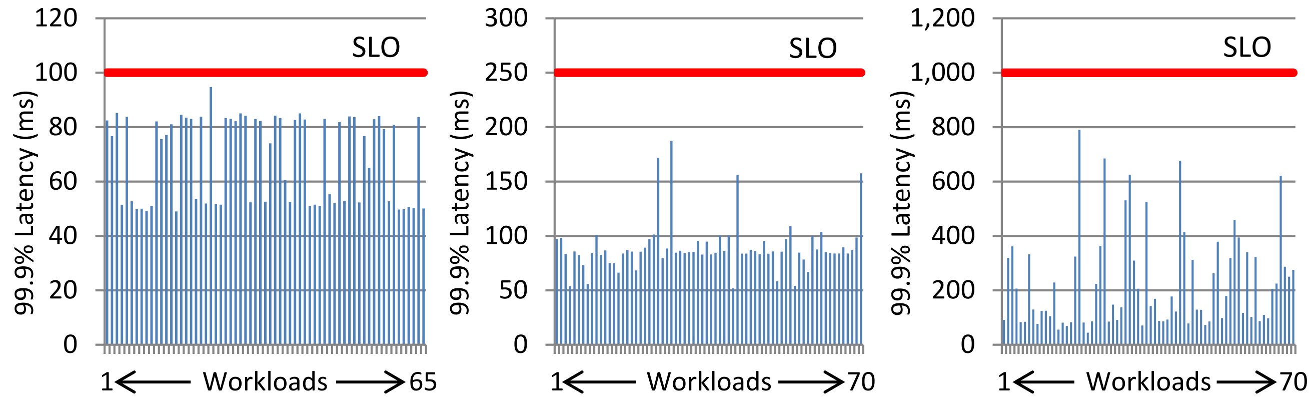


Figure 7: 99.9% latency (vertical bars) from running the "Same SLO" experiments in Fig. 6 on our cluster using WorkloadCompactor. All workload 99.9% latencies are below the SLO (horizontal line).

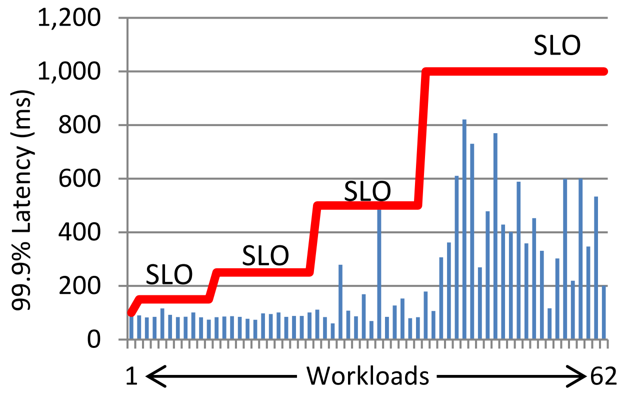


Figure 8: 99.9% latency from the "Random SLO" experiment in Fig. 6 with workloads grouped by SLO.

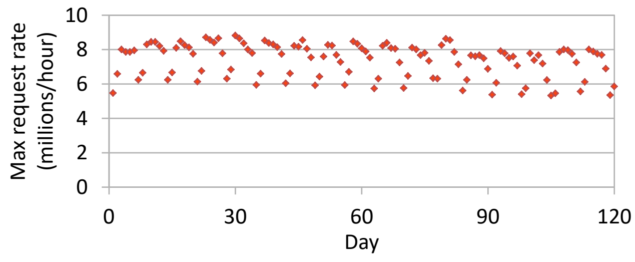


Figure 9: Each day's maximum hourly request rate for Wikipedia english pages, which is stable over months. The highest request rate is within 15% of the median day's maximum hourly request rate.

5.3 Scalability of workload packing density

Fig. 10 and Fig. 11 show the results from scaling the experiments from our local cluster experiments in Sec. 5.1 to more workloads. Our results show that WorkloadCompactor's packing density is not significantly affected by the size of the cluster, and we expect WorkloadCompactor to perform well regardless of the cluster size.

5.4 Scalability of computation

Fig. 12 shows the scalability of WorkloadCompactor's *computation* as the cluster size grows. Our results show that WorkloadCompactor's fast first-fit (FFF) policy (Sec. 2.4) scales much better than the typical naïve first-fit policy since FFF skips servers that are nearly full. Note that WorkloadCompactor's computation is not on the critical path for handling requests; it executes when new workloads arrive to the system.

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

5.5 Effect of workload departures

So far, we've assumed workloads only arrive over time. In reality, workloads will also depart from the system, leaving gaps in which to place future workloads. To mimic this behavior, we run an experiment where workloads randomly arrive and depart from the system. Our results in Fig. 13 show that WorkloadCompactor is better able to cope with workload departures than the state-of-the-art approaches, which use over 50% more servers. By contrast, the state-of-the-art approaches use over 40% more servers in the arrival-only scenario in Fig. 11. This is because WorkloadCompactor can dynamically reconfigure rate limits for previously placed workloads to better pack in new workloads, whereas the other approaches have less flexibility in squeezing in new workloads once a given workload has departed.

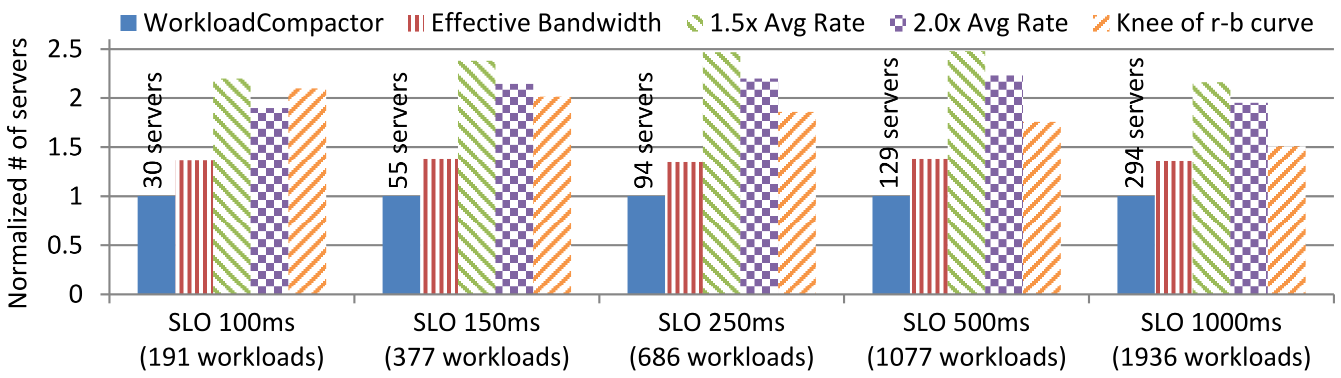


Figure 10: Scaling the Same SLO experiments in Fig. 6 to all available workloads. Since we only select workloads that can meet its SLO when run in isolation, there are more available workloads with higher SLOs. WorkloadCompactor continues to outperform the state-of-the-art approaches at larger scales.

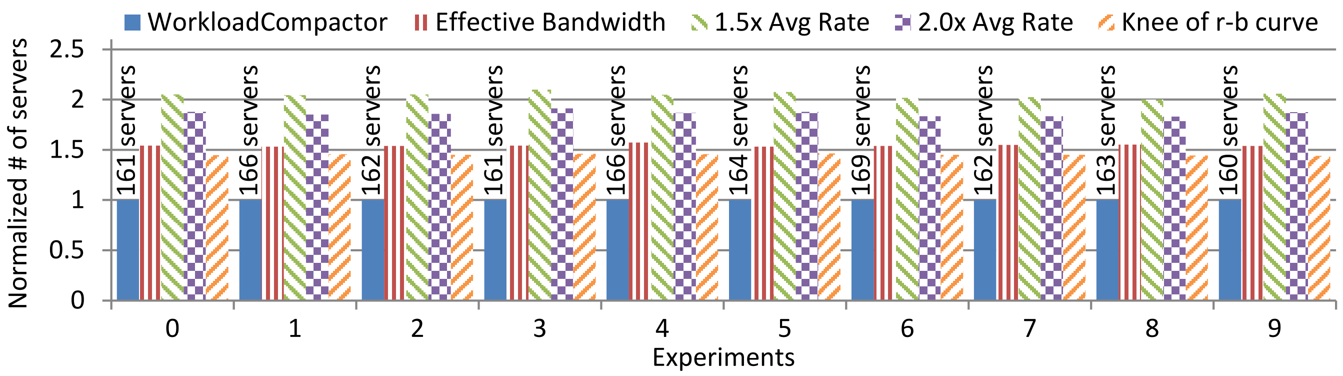


Figure 11: Scaling the Random SLO experiment in Fig. 6 to 1000 workloads. We repeat the experiment with ten random sets of 1000 workloads to show that WorkloadCompactor consistently outperforms state-of-the-art approaches.

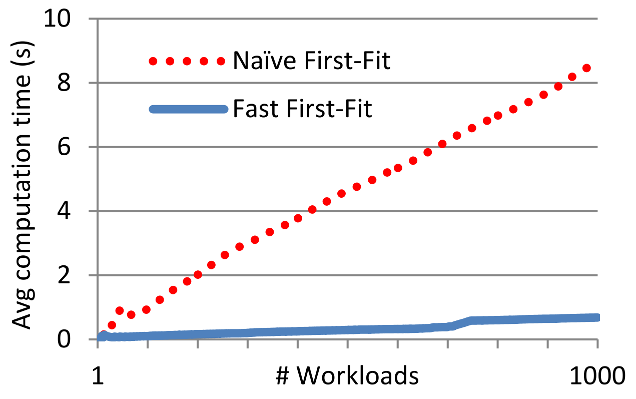


Figure 12: Comparing the computation time scalability of first-fit and WorkloadCompactor's fast first-fit (FFF) algorithm. FFF is much faster since it skips checking servers that are nearly full.

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

While storage is often a bottleneck, the network can also become a bottleneck depending on the number and bandwidth of SSDs vs. the network bandwidth. Fig. 14 shows an experiment where we vary the number of SSDs per server to demonstrate this effect. When storage is a bottleneck, increasing the number of SSDs per server should decrease the number of servers used. Eventually, adding more SSDs per server does not help, since now the network has become the bottleneck. For example, in our system, we see that storage is a bottleneck with a single SSD per server, but the network becomes a bottleneck with 2+ SSDs per server. With 2+ SSDs per server, the number of servers used plateaus at around 115 servers, and the number of SSDs used also plateaus since the extra SSDs aren't helpful. In systems with higher network bandwidth, we would expect similar trends, except with the plateau occurring at a higher number of SSDs per server. Importantly, WorkloadCompactor is designed to account for both storage and network, and it will pack workloads so as to not overload either storage or network.

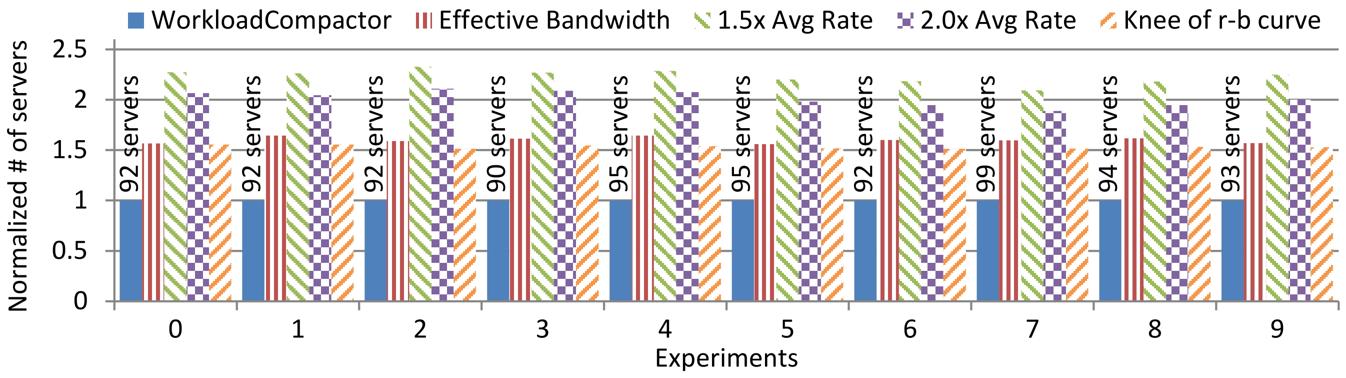


Figure 13: Same experiment as Fig. 11, except with workloads randomly arriving and departing over time. Results measure the maximum number of servers used at any point in time, normalized to WorkloadCompactor. Comparing results to Fig. 11, we see that WorkloadCompactor handles workload departures better than other approaches since WorkloadCompactor's dynamic reconfiguration naturally adapts to departures.

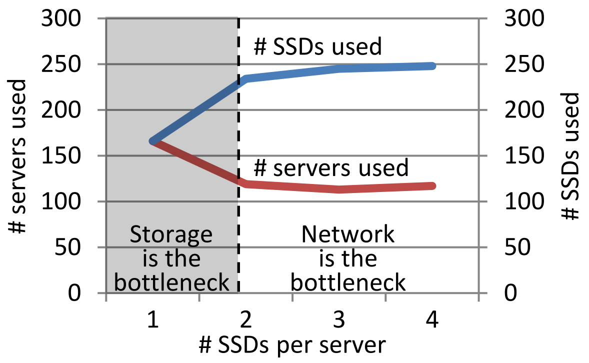


Figure 14: The effect of changing the number of SSDs per server in an experiment with 1000 random workloads, each with random SLOs. With 1 SSD per server, the storage is a bottleneck. With 2+ SSDs per server, the network becomes a bottleneck, causing the number of servers and number of SSDs used to plateau. Since WorkloadCompactor accounts for both network and storage, it naturally detects that it doesn't need to use the extra SSDs per server since the network is fully loaded.

5.7 Multiple simultaneous rate limits

In addition to the state-of-the-art approaches for selecting rate limits, there is another approach proposed in PriorityMeister [26] to use multiple rate limiters simultaneously for each stage (e.g., storage, network) in a workload. Ideally, using multiple simultaneous rate limits will achieve a similar benefit to dynamically reconfiguring rate limits, but there are multiple caveats. First, enforcing multiple simultaneous rate limits is uncommon in systems today, making it harder to deploy. Second, the complexity in analyzing tail latency with multiple rate limits leads to 15× more computation time than WorkloadCompactor with 1000 workloads. Third, the complexity also leads to the analysis being overly conservative when handling

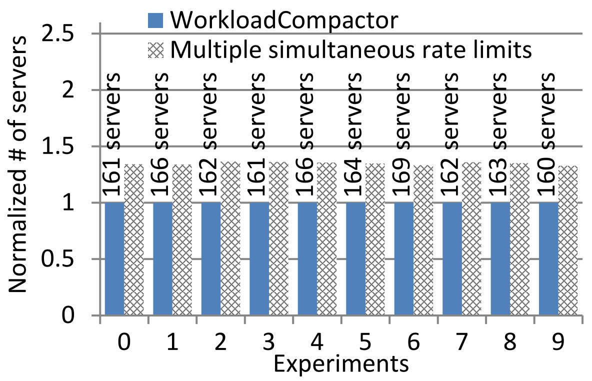


Figure 15: Comparing WorkloadCompactor to using multiple simultaneous rate limits.

equal priority workloads, which results in using more servers than necessary. Consequently, WorkloadCompactor uses fewer servers than the multiple simultaneous rate limits approach, as seen in Fig. 15.

6 RELATED WORK

WorkloadCompactor is related to three branches of work, and is the first system to address all three areas. WorkloadCompactor solves the *workload placement* problem in the context of *meeting tail latency SLOs* by optimizing the *selection of rate limit parameters*. Tbl. 1 summarizes the differences between related works.

Workload placement

There are many works that consider how to place and migrate workloads between servers [8, 10, 12, 16, 18]. Many of these works propose good ideas for how to improve latency and throughput with better load balancing [10, 12, 16, 18]. However, ensuring that tail latency SLOs are met is outside the scope of their work.

Delphi/Pythia [8] looks at migrating workloads to meet tail latency SLOs. It reacts to SLO violations and learns the appropriate

		Workload placement	Tail latency SLOs	Rate limit configuration
Load balancing and migration	Basil [10]	✓	✗	✗
	Pesto [12]	✓	✗	✗
	Romano [16]	✓	✗	✗
	VectorDot [18]	✓	✗	✗
	Delphi/Pythia [8]	✓	✓	✗
Guaranteeing tail latency SLOs	Silo [13]	✓	✓	✗
	QJump [9]	✗	✓	✗
	PriorityMeister [26]	✗	✓	✓
	SNC-Meister [25]	✗	✓	✗
Rate limit configuration	Effective bandwidth [15]	✗	✓	✓
	WorkloadCompactor	✓	✓	✓

Table 1: Comparison of related work.

mitigation actions (e.g., which tenant to migrate). A major limitation is that at the core of its design, it allows SLO violations to occur and then reacts. By contrast, WorkloadCompactor is designed to avoid SLO violations rather than fix bad placements.

Tail latency SLOs

There are four systems that provide tail latency SLO guarantees: Silo [13], QJump [9], PriorityMeister [26], and SNC-Meister [25]. Like WorkloadCompactor, they all use mathematical analysis to ensure SLOs can be met.

Of these works, Silo is the only system that addresses workload placement. The authors find that a first-fit policy works well to pack workloads onto servers. However, Silo does not address how to set rate limits, and the key finding in our work is that the choice of rate limits significantly impacts the ability to compact workloads onto servers. In addition, Silo (as well as QJump and SNC-Meister) only support networks, whereas WorkloadCompactor supports both storage and networks. Furthermore, WorkloadCompactor also introduces the fast first-fit feature that drastically improves the computational scalability of workload placement (see Sec. 5.4).

Of these works, PriorityMeister is the only system that considers how to select rate limits. PriorityMeister introduces the idea of simultaneously using multiple rate limiters to avoid picking a specific $\langle r, b\rangle$ rate limit. Conceptually, the idea should work well, but as described in Sec. 5.7, there are multiple caveats that make WorkloadCompactor a superior solution. Additionally, workload placement is outside the scope of the PriorityMeister work.

Selection of rate limit parameters

Little is known about selecting rate limit parameters since most works using rate limits (e.g., [9, 11, 13, 24]) assume the user is responsible for selecting them. Users end up relying upon ad hoc heuristics such as scaling the average rate by a factor [13]. The state-of-the-art from theory is an idea known as effective bandwidth [15], described in Sec. 4.1. Though effective bandwidth is optimal when workloads have the same SLO and only traverse a single stage (e.g., storage), our experiments show that WorkloadCompactor uses far fewer servers than effective bandwidth when handling multiple stages or workloads with different SLOs.

7 CONCLUSION

WorkloadCompactor is a new system for compacting workloads onto servers while meeting tail latency SLOs. To guarantee tail latency SLOs, WorkloadCompactor enforces rate limits and priorities in storage and network and uses network calculus equations to check if workloads can be placed together while meeting their SLOs. Surprisingly, we find that the selection of workload rate limits makes a big difference in the ability to pack workloads together. In a sense, a workload's rate limit defines the "size" of the workload and affects how well it fits on a server. Rate limits have both a rate (r) and burst (b) parameter, and there is flexibility in trading off r and b. WorkloadCompactor takes advantage of this flexibility to better pack workloads onto servers as workloads arrive and depart. WorkloadCompactor's new technique for optimizing the selection of rate limits can compact more workloads onto a server while meeting SLOs. Our compaction technique is used in conjunction with our scalable placement algorithm, which places workloads onto servers an order of magnitude faster than the traditional first-fit policy. Experiments with assigning 1000 workloads to servers show that WorkloadCompactor is superior to state-of-the-art approaches, which use 40-150% more servers than WorkloadCompactor.

ACKNOWLEDGMENTS

This research is supported in part by Intel as part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC), by a Google Faculty Research Award 2015/16, by a Facebook Faculty Research Award 2015/16, and by the National Science Foundation under awards CMMI-1538204, CMMI-1334194, CSR-1116282, and XPS-1629444. We also thank the member companies of the PDL Consortium for their interest, insights, feedback, and support.

REFERENCES

[1] 2017. Wikimedia Downloads: Analytics Datasets. (2017). https://dumps.wikimedia.org/other/analytics/

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

[3] Yossi Azar, Ilan Reuven Cohen, Seny Kamara, and Bruce Shepherd. 2013. Tight Bounds for Online Vector Bin Packing. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing (STOC '13)*. ACM, New York, NY, USA, 961–970. https://doi.org/10.1145/2488608.2488730

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

[5] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. https://doi.org/10.1145/2408776.2408794

[6] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *ACM SOSP*. 205–220.

[7] Cagdas Dirik and Bruce Jacob. 2009. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. In *ACM ISCA*, Vol. 37. 279–289.

[8] Aaron J. Elmore, Sudipto Das, Alexander Pucher, Divyakant Agrawal, Amr El Abbadi, and Xifeng Yan. 2013. Characterizing Tenant Behavior for Placement and Crisis Mitigation in Multitenant DBMSs. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 517–528. https://doi.org/10.1145/2463676.2465308

[9] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. 2015. Queues Don't Matter When You Can JUMP Them!. In *USENIX NSDI*.

[10] Ajay Gulati, Chethan Kumar, Irfan Ahmad, and Karan Kumar. 2010. BASIL: Automated IO Load Balancing Across Storage Devices. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*. USENIX Association, Berkeley, CA, USA, 13–13. http://dl.acm.org/citation.cfm?id=1855511.1855524

[11] Ajay Gulati, Arif Merchant, and Peter J. Varman. 2007. pClock: an arrival curve based approach for QoS guarantees in shared storage systems. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems (SIGMETRICS '07)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/1254882.1254885

[12] Ajay Gulati, Ganesha Shanmuganathan, Irfan Ahmad, Carl Waldspurger, and Mustafa Uysal. 2011. Pesto: Online Storage Performance Management in Virtualized Datacenters. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing (SOCC '11)*. ACM, New York, NY, USA, Article 19, 14 pages. https://doi.org/10.1145/2038916.2038935

[13] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable Message Latency in the Cloud. In *ACM SIGCOMM*. ACM, 435–448.

[14] Swaroop Kavalanekar, Bruce L. Worthington, Qi Zhang, and Vishal Sharda. 2008. Characterization of storage workload traces from production Windows Servers.. In *IISWC* (2008-10-29), David Christie, Alan Lee, Onur Mutlu, and Benjamin G. Zorn (Eds.). IEEE, 119–128. http://dblp.uni-trier.de/db/conf/iiswc/iiswc2008.html#KavalanekarWZS08

[15] Jean-Yves Le Boudec and Patrick Thiran. 2001. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet.* Springer-Verlag, Berlin, Heidelberg.

[16] Nohhyun Park, Irfan Ahmad, and David J. Lilja. 2012. Romano: Autonomous Storage Management Using Performance Prediction in Multi-tenant Datacenters. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12)*. ACM, New York, NY, USA, Article 21, 14 pages. https://doi.org/10.1145/2391229.2391250

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

[18] Aameek Singh, Madhukar Korupolu, and Dushmanta Mohapatra. 2008. Server-storage Virtualization: Integration and Load Balancing in Data Centers. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC '08)*. IEEE Press, Piscataway, NJ, USA, Article 53, 12 pages. http://dl.acm.org/citation.cfm?id=1413370.1413424

[19] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. 2013. IOFlow: A Software-defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 182–196. https://doi.org/10.1145/2517349.2522723

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

[21] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. 2013. Bobtail: Avoiding Long Tails in the Cloud. In *USENIX NSDI*. 329–342.

[22] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 15–28. https://www.usenix.org/conference/fast17/technical-sessions/presentation/yan

[23] Young Jin Yu, Dong In Shin, Hyeonsang Eom, and Heon Young Yeom. 2010. NCQ vs. I/O Scheduler: Preventing Unexpected Misbehaviors. *ACM Trans. Storage* 6, 1 (April 2010), 2:1–2:37.

[24] Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel. 2006. Storage performance virtualization via throughput and latency control. *Trans. Storage* 2, 3 (Aug. 2006), 283–308. https://doi.org/10.1145/1168910.1168913

[25] Timothy Zhu, Daniel S. Berger, and Mor Harchol-Balter. 2016. SNC-Meister: Admitting More Tenants with Tail Latency SLOs. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*. ACM, New York, NY, USA, 374–387. https://doi.org/10.1145/2987550.2987585

[26] Timothy Zhu, Alexey Tumanov, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. 2014. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *ACM SOCC*. ACM, New York, NY, USA, Article 29, 14 pages. https://doi.org/10.1145/2670979.2671008