Next Generation Cloud Computing Architectures: Performance and Pricing

Kunal Mahajan

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2021

Abstract

Next Generation Cloud Computing Architectures: Performance and Pricing

Kunal Mahajan

Cloud providers need to optimize the container deployments to efficiently utilize their network, compute and storage resources. In addition, they require an attractive pricing strategy for the compute services like containers, virtual machines, and serverless computing in order to attract users, maximize their profits and achieve a desired utilization of their resources. This thesis aims to tackle the twofold challenge of achieving high performance in container deployments and identifying the pricing for compute services.

For performance, the thesis presents a transport-adaptive network architecture (D-TAIL) improving tail latencies. Existing transport protocols such as Homa, pFabric [1, 2] utilize Shortest Remaining Processing Time (SRPT) scheduling policy which is known to have starvation issues for long flows as SRPT prioritizes short flows. D-TAIL addresses this limitation by taking age of the flow in consideration while deciding the priority. D-TAIL shows a maximum reduction of 72%, 29.66% and 28.39% in 99th-percentile FCT for transport protocols like DCTCP, pFabric and Homa respectively. In addition, the thesis also presents a container deployment design utilizing peer-to-peer network and virtual file system with content-addressable storage to address the problem of cold starts in existing container deployment systems. The proposed deployment design increases compute availability, reduces storage requirement and prevents network bottlenecks.

For pricing, the thesis studies the tradeoffs between serverless computing (SC) and traditional cloud computing (virtual machine, VM) using realistic cost models, queueing theoretic performance models, and a game theoretic formulation. For customers, we identify their workload distribution between SC and VM to minimize their cost while maintaining a particular performance constraint. For cloud provider, we identify the SC and VM prices to maximize its profit. The

main result is the identification and characterization of three optimal operational regimes for both customers and the provider, that leverage either SC or VM only, or both, in a hybrid configuration.

Table of Contents

List of Tables

List of Figures

Acknowledgements

Undertaking this PhD has been a truly life-changing experience for me and it would not have been possible to do without the support and guidance that I received from many people.

I will forever be thankful to my advisors, Professors Vishal Misra and Dan Rubenstein, for all the support and encouragement they gave me during the last five years. Their guidance and constant feedback was instrumental in achieving this PhD. They have been very supportive and have given me the freedom to pursue various projects in different fields. Vishal and Dan are the smartest people I know. In addition to many insightful discussions, our group meetings also had political discussions, Vishal's witty remarks, sports talk, thereby creating the best environment for personal and professional development. Dan always said,

"The paper should tell a story."

These are the very words that I live by, not just for scientific contributions but also for everyday things. I am still learning new things every day and I try to imbibe the invaluable advice I received from my advisors.

I am very grateful to Professor Javad Ghaderi for his scientific advice and knowledge and many insightful discussions and suggestions, during our collaboration in the first phase of my PhD. I would also like to thank my former research advisor, Professor Steven Nowick, for guiding me when I joined Columbia.

Thank you to Professor Daniel Figueiredo from UFRJ, Brazil. It was a delight collaborating with you on our Serverless Computing work.

Special thanks to Dr. Edin Kadric, who has been like my mentor since we met during my undergrad at University of Pennsylvania.

I would also like to thank my colleagues in DNA lab: Kevin Yang and Niloofar Bayat. It was a great experience to collaborate with them on projects. Thank you to all my colleagues and

collaborators with whom I have discussed ideas and worked together: Alex Stein, Pearce Kieser, Mehrnoosh Shafiee, Tingjun Chen, Todd Arnold, Craig Gutterman, and Saket Mahajan.

Thank you also to the staff of the Department of Computer Science, Jessica Rosa, Oana Moldoveanu, Cindy Meekins, the CRF team, and many others for their help during my PhD program.

Finally, I would like to thank my parents and family, for their constant support, encouragement and being the strong pillars in my life.

Dedication

Dedicated to my dad, Vasudeo Mahajan, and my mom, Kiran Mahajan, for their unending love and guidance without which the PhD would not have been achievable.

Introduction

Cloud computing is a disruptive technology that over the past two decades has dramatically changed how enterprises compute and how computation is monetized. The enterprises are increasingly deploying their applications on the cloud infrastructure provided by major IT corporations such as Amazon, Google, and Microsoft. One of the main driving forces behind this transformation is the large benefit for both the cloud service providers and the enterprises. The cloud services represents a major source of profit for the cloud providers: in 2017, profit with Amazon's AWS Cloud Services surpassed the profit of all other Amazon sales combined [3]. In fact, Microsoft reported revenue increase of 35% for its cloud division Azure even during the times of the Covid-19 pandemic [4]. For enterprises, cloud services offer the ability to deploy applications quickly, minimize the monetary costs compared to private cloud, reliability and scalability.

Cloud computing platforms offer a plethora of computing paradigms like Virtual Machines (VM), Containers and Serverless Computing. The market competition forces the cloud providers to ensure high performance of their computing paradigms and attractive pricing strategy of these services to garner market share. For performance, the cloud provider needs to optimize their three resources: networks, compute and storage. While research has improved the performance of these three resources, the current approaches have significant limitations that lower the performance. For instance, in datacenter networks, research has provided new transport protocols utilizing prioritization of flows to minimize flow completion times. These protocols prioritize small flows and degrade the performance of large flows, potentially starving them [1, 2]. In compute and storage, the cloud architecture of deploying containers suffers from the problem of cold start. This thesis addresses these limitations and provides performance improvements in network, compute and storage over existing work. Finally, the thesis also provides an attractive pricing strategy for the computing paradigms for the cloud providers to ensure maximum profit assuming smart users who

aim to minimize their monetary costs.

In Chapter 2, we focus on improving the performance of datacenter networks. We present D-TAIL, Datacenter Transport Adaptive Improvement in Latencies, an adaptive mechanism easily implemented on top of existing datacenter transport protocols such as Homa, pFabric, and DCTCP to improve targeted performance metrics, and in this work we focus on improving tail latencies. The existing protocols are designed to reduce delays for high-volume short flows, but increase tail latencies due to significant starvation of larger flows, reducing large flow throughput by an order of magnitude. Our mechanism works in conjunction with the underlying transport protocol to maintain low delays for short flows while addressing starvation of the larger flows. D-TAIL can be used in environments where flow size information is not known prior to transmission (size-blind), and is further improved when flow size is known (size-aware). The design utilizes a generalized, adaptive scheduler that, using a single parameter, can be tuned to implement a range of conventional scheduling policies, including SRPT, FCFS, LAS, as well as hybrids of these policies. Bayesian optimization is used to identify parameter values that address the datacenter operator's metric of interest. Our implementation in simulation and upon a Cloudlab testbed demonstrates D-TAIL's efficacy in controlling tail latencies and optimizing flow completion times across a range of well-known data center workloads. We highlight the importance of the adaptability of our mechanism with experiments that demonstrate improved performance on different workloads over classical mechanisms like SRPT or LAS. We show that D-TAIL's size-blind and size-aware variants reduce DCTCP's 99th-percentile flow completion time (FCT) by up to 50.14% and 72% respectively in realistic datacenter workloads. We further show a maximum reduction of 29.66% and 28.39% in 99th-percentile FCT for optimized transport protocols like pFabric and Homa respectively.

In Chapter 3, we focus on addressing the cold start problem in container deployments by exploiting content similarity. The cold start problem arises due to the current container deployment design, which requires retrieving the source code from storage to a worker node, creating the container with source code and executing the container. This process is executed for each new instance of the container. The problem is exacerbated with concurrency, thereby impacting

application performance. To solve the cold start problem, we analyze current modes of deployment and identify data similarities across applications. Based on these observations, we propose a new container deployment system that is built atop a peer-to-peer network, virtual file-system and content-addressable storage, which will increase compute availability, reduce storage requirement, and prevent network bottlenecks.

In Chapter 4, we study the problem of identifying optimal pricing for the computing paradigms: Virtual Machines and Serverless Computing. We present an analysis of the potential cost benefits of serverless computing for end customers and cloud providers. Using realistic cost models, queueing theoretic performance models, and a game theoretic formulation, we explore the tradeoffs between serverless computing (SC) and traditional cloud computing (virtual machine, VM). In the proposed framework, customers distribute their workload between SC and VM to minimize their cost while maintaining a particular performance constraint, while the cloud provider sets SC and VM prices to maximize its profit. We explore the impact of relative prices, customer workload, service capacity, and provider costs. Our main result is the identification and characterization of three optimal operational regimes for both customers and the provider that leverage either SC or VM only, or both, in a hybrid configuration. The various insights provided in this chapter can help both cloud providers and customers better understand the tradeoffs and implications of a hybrid system that combines serverless and VM rental with corresponding pricing models.

Chapter 2 is done in collaboration with Pearce Kieser, Alex Stein, Yudong Yang, Vishal Misra, Dan Rubenstein, and is submitted to the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2021). Chapter 3 is done in collaboration with Saket Mahajan, Vishal Misra, Dan Rubenstein and is published in the Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies (CoNEXT 2019) [5]. Chapter 4 is done in collaboration with Daniel Figueiredo, Vishal Misra, Dan Rubenstein and is published in the IEEE Global Communications Conference (GLOBECOM 2019) [6].

Chapter 2: D-TAIL: Improving tail performance of smart schedulers in the datacenter

The global data center market is expected to grow at an accelerated compounded annual rate of over 10% until 2022, with an estimated market value increase of $96.99 billion [7]. The significant growth has fueled the development of an efficient network stack, new transport protocols, and new congestion control mechanisms. The primary aim of these advances has been to achieve low latency for short flows and high throughput for large flows, with an emphasis on minimizing tail latency as datacenters use tail latency as a key metric to measure their performance [1, 2, 8, 9, 10, 11]. The transport protocols achieve lower tail latencies by prioritizing short flows that degrades the performance of large flows, potentially starving them.

In this chapter, we design a low (tail) latency network architecture which prevents starvation of large flows and leads to lower tail latencies. The design utilizes and is compatible with existing transport sender-driven and receiver-driven protocols like Homa, pFabric, DCTCP [1, 2, 9]. It is information adaptive, i.e., it can utilize flow size information when it is available, but operate effectively without this information in case it is not available.

2.1 Introduction

The massive parallelism provided to applications by large-scale distributed systems and microservices architectures can make response times, if not appropriately controlled, unpredictable. The variability in latencies can be due to various factors such as resource sharing, energy management, and load imbalance [12, 13]. Recent works, such as Homa [1] and pFabric [2], have shown 100x lower latencies than the best published measurements by prioritizing the short-lived flows. Shortest Remaining Processing Time (SRPT), the policy utilized by Homa and pFabric, is known

to be the optimal policy in terms of optimizing mean response times [1, 2, 14].

However, SRPT does not consider the sojourn time or age of a job, and hence is known to starve the larger jobs of workloads. Prior work that reduced average latency exhibits an exponential rise in the 99th-percentile slowdown, with larger (tail) flow completion times (FCT) increasing by up to a factor of 30 [1]. Ideally, the network architectures would achieve low mean latencies without inducing starvation of larger flows, regardless of the workload. Our work addresses this issue: How can tail latency for existing protocols be managed easily and immediately without much impact on the (non-tail) flows?

Additionally, some transport mechanisms, such as Homa and pFabric assume that the underlying flow sizes are known at the time a flow is scheduled, as knowledge of flow size information substantially improve data center network performance [1, 2]. As studied in [15], flow size information can be estimated to a accuracy greater than 80% for Web Search, TensorFlow, and PageRank workloads. However, [15] also states that it is not plausible to have flow size information accurately available for other workloads such as Stochastic Gradient Descent (SGD) [16]. In particular, data centers who host third party customers can rarely assume flow size information is known in advance. In short, network architectures should be adaptive to utilize flow size information, if available, for scheduling, but should remain functional when it is not available.

In this chapter, our goal is to design a low (tail) latency network architecture with the following properties/characteristics:

- *Reuse of well-tested protocols*: Utilize and be compatible with existing transport sender-driven, and receiver-driven protocols like Homa, pFabric, and DCTCP.
- *Information Adaptive*: Utilize flow size information (size-aware mode) when it is available, but operate effectively without this information in case it is not available (size-blind mode).
- *Avoid starvation*: Limit the 99th-percentile FCT for large flows.
- *Robustness*: Work with a variety of workload types and distributions (RPC style or Map Reduce, heavy tailed/light tailed), traffic characteristics (incast, other kinds) and data center

topologies.

We explore an approach that attempts to achieve the low tail latency without significantly compromising delays of short-lived flows, with minimal changes to data center architectures as they exist today. Our insight is to use an elegant scheduling policy that "revives" or re-prioritizes long-lived flows that are starving, such that the starved flows are completed with a gradual revival, receiving their needed service in pieces such that short lived flows are not significantly delayed. We propose D-TAIL, Datacenter Transport Adaptive Improvement in Latencies, an enhancement mechanism, which can be added to a variety of existing protocols such as Homa, pFabric, and DCTCP [1, 2, 9]. D-TAIL uses a priority-based scheduling mechanism that is flexible and can operate without flow size information (size-blind mode), but is further enhanced by using flow size information (size-aware mode) when available. The generalized scheduling mechanism can be dynamically adapted to implement a range of scheduling policies like SRPT, FCFS, PS, LAS [14, 17], as well as hybrid policies that best fit the needs of current datacenter workload using a single tunable parameter to adapt the policy. Bayesian optimization is used to identify the value of tunable parameter to meet the performance objectives of each workload. While our mechanism can work with any performance objective, in this chapter we focus on improving tail latencies.

The chapter makes four major contributions:

- We design a configurable, blind scheduler for datacenter networks, via basic adaptation of PBS [17], for packet scheduling to the networking context.
- We extend the configurable blind scheduler into a configurable size-aware scheduler, that utilizes flow size information.
- We show how to extend existing protocols to include the D-TAIL mechanism using only the information and hardware capabilities assumed by the respective underlying protocol.
- We present both simulation as well as an experimental testbed results on cloudlab[18], demonstrating that we can maintain mean FCT and significantly reduce tail latencies for

a variety of real data center workloads. For instance, size-aware reduced Homa's mean FCT and 99th%-tile FCT by 22.12% and 28.39% respectively in Facebook Hadoop workload.

The remainder of this chapter is organized as follows. §2.2 presents the motivation and key ideas behind D-TAIL. §2.3 provides detailed description of the D-TAIL design, priority computation, and transport protocol-specific implementations. Extensive experimental results of packet-level simulations are shown in §2.4. §2.5 details our D-TAIL implementation in physical testbed and testbed results. §2.6 discusses and mentions future work, followed by a summary of related work in §2.7.

2.2 D-TAIL Motivation and Key Ideas

This section discusses the key ideas behind the design of D-TAIL. Before we delve into the detailed design, we explain why we believe using existing transport protocols is important. Many recent data center architectures with newer transport protocols such as Homa, NDP, and pFabric, have shown significant performance gains over the existing data center deployment [1, 8, 2, 11]. The implementation of these existing protocols affects the entire data center industry including the chip designers, chip manufacturers, switch manufacturers, and data center operators [19]. This requires massive investment in terms of time, research, development, and money. Therefore, it is essential to enhance existing, high-performance architectures with minimal software modifications.

Utilizing existing transport protocols Existing transport protocols are well-studied and have been fine-tuned to perform effectively across a wide range of data center environments [1, 2, 20, 21]. They have been deployed worldwide and data center operators are well-versed in managing and troubleshooting. In addition, ideally, we wish to roll out changes in ways which minimally disrupt users. Requiring application programmers to consider problems in the space traditionally managed by transport protocols is an additional complexity unlikely to be welcomed by the community. To support existing transport protocols when making such changes enables the code reuse and the portability of engineering expertise necessary for speedy deployment.

Tunable scheduling policy The workloads in the data center are not predictable; they are likely

to change over time as new applications are developed and different types of users arise. The scheduling policy used in the data center thus should not be based solely on past workload traces, but should allow the data center administrator to decide how flows within the workload should be prioritized. This drives our design: using D-TAIL, the data center administrator can specify the performance objectives (for e.g. 99th-percentile FCT, 99th-percentile slowdown, mean FCT) and identify the parameter value for the desired performance objective using Bayesian optimization.

2.2.1 PBS Background and Extensions

The motivation of the chapter comes from observing, adapting and extending a generalized blind scheduling policy called PBS[17]. Scheduling is a compromise not only between individual tasks, but also between systems with different workload patterns, between different performance requirements, including mean response time, mean slowdown, responsiveness and fairness measures. PBS first introduced a generalized blind flexible scheduling policy to balance these requirements. Specifically, if a task i has sojourn time as $t_i(t)$ and attained service $x_i(t)$, then at a high level a scheduling policy defines fairness in one of two traditional ways:

- Seniority — Prioritize larger sojourn time $t_i(t)$
- Service requirements — Prioritize smaller attained service $x_i(t)$

PBS proposed a balance of these two kinds of fairness requirements by proposing a policy which schedules a task with maximal $t_i(t) - \alpha x_i(t)$ where α is a configurable parameter in $[0, \infty)$. More generally: $g(t_i(t)) - \alpha g(x_i(t))$, where $g(\cdot)$ is some function, e,g., $\log t_i(t) - \alpha \log x_i(t)$. The PBS policy is for every task i, compute its **priority value** $p_i(t) = \log t_i(t) - \alpha \log x_i(t)$, equivalent to $P_i(t) = \frac{t_i(x)}{[x_i(t)]^\alpha}$, and at time t, serve the task with the highest priority P_i (or processor share if two jobs have the same priority level). α is the single tunable parameter which lets the policy move between FCFS ($\alpha = 0$), Generalized PS ($\alpha = 1$), Least Attained Service ($\alpha = \infty$) as well as hybrid schedules policies where $0 < \alpha < 1$ or $1 < \alpha < \infty$, see [17] for details.

Our Size-Aware Extension to PBS

We **extend** PBS to cover the case where job size information is available, a scenario common for a set of data center workloads. We simply replace attained service, $x_i(t)$, in the definition of PBS with $r_i(t)$: the remaining job size at time t, which allows us to design a flexible policy that can adapt to different workloads and performance metrics. When the full job size information is available, the Shortest Remaining Processing Time (SRPT) is known to be the optimal policy in terms of optimizing mean response times. With this simple but important extension to PBS, we obtain a flexible size-aware policy which can balance requirements between job size and seniority fairness. Similar to blind PBS, size-aware PBS moves between FCFS ($\alpha = 0$), enhanced SRPT ($\alpha = \infty$) as described in §2.2.1 and hybrids in between.

Overcoming SRPT limitations

SRPT does not consider the sojourn time or age of a job, and hence is known to have starvation issues with long running jobs [14]. D-TAIL overcomes this limitation of SRPT by also incorporating the age of the flow into the priority metric. The priority value of a starved long flow will increase as the age of flow increases; thereby, allowing even flows with large remaining job sizes to be served in short bursts, progressing its transmission without much impact on the small flows. This key contribution is shown in §2.4.2.

2.3 D-TAIL Design

In this section, we describe the implementation of D-TAIL on top of existing protocols such as sender-driven protocols (DCTCP, pFabric) and receiver-driven protocols (Homa). In general, D-TAIL's design consists of modules like per-flow priority computation using a tunable parameter (§2.3.1), priority quantization (§2.3.2) and priority-based scheduling for flows. Generally, the different underlying scheduling algorithms make varying assumptions about the availability of information (e.g., flow sizes), as well as the capabilities of components of the datacenter architec-

ture. When D-TAIL is used to enhance a particular underlying protocol, the D-TAIL implementation respects and can use the information, capabilities and components assumed by the underlying protocol that it is enhancing.

2.3.1 Priority computation

As mentioned in §2.2.1 and §2.2.1, we now have two scheduling policies available for flows: size-blind and size-aware. We extend these two policies in datacenter networks in D-TAIL. Specifically, we use the basic PBS mechanism to decide the priority of a flow in the size-blind mode, where the sojourn time of the job is simply the age of the flow, and attained service is the number of bytes transmitted.

$$P_i(\text{size-blind}) = \frac{\text{age of flow}}{(\text{sent bytes})^{\alpha}} \tag{2.1}$$

In the size-aware mode, the sojourn time is still the age of the flow while we replace *sent bytes* with *remaining bytes* in equation 2.1 to compute priority of the packet, as shown below.

$$P_i(\text{size-aware}) = \frac{\text{age of flow}}{(\text{remaining bytes})^{\alpha}} \tag{2.2}$$

The age of flow is measured in nanoseconds. We record the time when the first packet of each flow is scheduled and compute the age of flow for other packets in the flow by simply finding the time difference in nanoseconds. The sent bytes includes the data bytes transmitted by the sender, and remaining bytes is simply flow size - sent bytes.

The numerator serves to linearly increase the priority of flows that may be experiencing starvation, while the denominator serves to gracefully demote flow priority as more service is attained (size-blind mode) or increase priority as a flow nears completion (size-aware mode). The speed of this change is determined by α. In our experimental section, we will demonstrate the importance of having a tunable α which adapts to a variety of workload distributions.

2.3.2 Priority Quantization

D-TAIL is governed by an equation that generates continuous priority values. When switches have the ability to support arbitrary priority values, as is the case in pFabric, the priority values can be used directly. However, most of the currently available commodity switches support a limited number of (typically four or eight) priority classes. Therefore we need an additional process for mapping these values into finite priority classes. Homa [1] introduced an elegant mechanism to map the continuous range of priority values into a finite number of priority buckets, and we replicate the mechanism in our implementation.

As in the Homa work, we observe priority values computed on the fly for each workload distribution for one particular alpha. From these priority values we choose boundaries which attempt to evenly distribute packets across the discrete priority classes. In the blind version of D-TAIL, the first packet of each flow is sent in the highest priority class (when both numerator and denominator are 0 and hence the metric is not well-defined). Higher priority values are sent to higher priority classes. Similarly for size-aware mode, even though the metric is 0 as the numerator is 0, the first packet of each flow is still sent in the higher priority class. This is done to avoid potential starvation of the flow since 0 is the lowest priority possible.

2.3.3 Transport Protocol-Specific Modifications

To enhance existing transport protocols, D-TAIL needs to dynamically track per-flow information, specifically bytes remaining and flow age. Figure 2.1 presents the design of D-TAIL, describing the flow of packet starting at the end-host and propagating through the switches. The end-host, which can be the sender or receiver depending on the protocol, requires a flow information table, packet header parser, and packet classifier. The packet header parser obtains the 4-tuple flow information from each packet and stores the flow metainfo in the table. For size-aware mode, the flow metainfo also includes the flow size which is included in the packet header in our implementation. The packet then arrives at the classifier that follows a 4-step process to compute the priority class for the packet. The first step is a high priority filter which assigns the first packet of

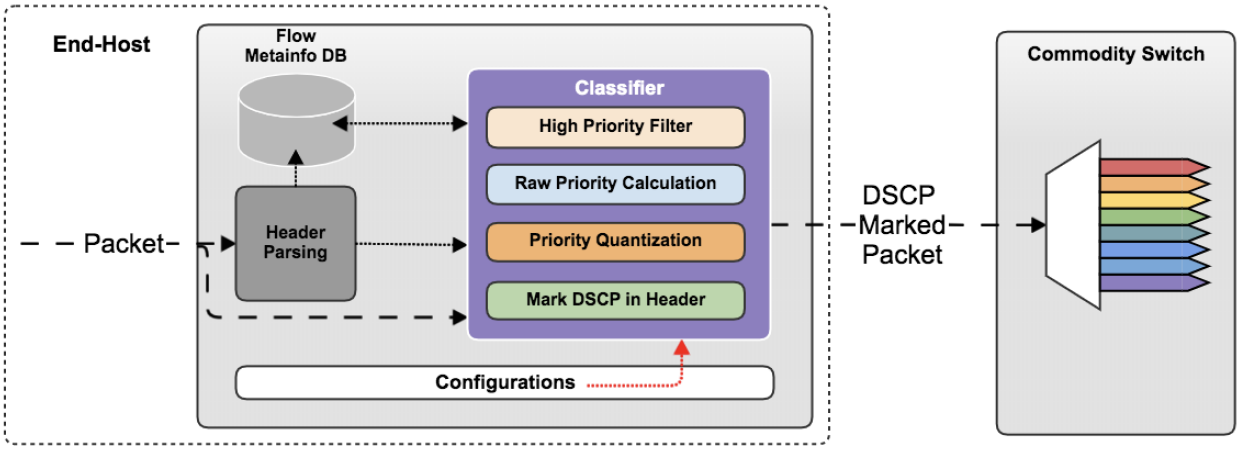


Figure 2.1: D-TAIL design

any flow to the highest priority class. For subsequent packets, priority value is computed for each packet, more details in §2.3.1, followed by a quantization method to determine the priority class explained in §2.3.2. The sender then marks the priority in DSCP bits in the packet header. The DSCP marked packet is transmitted to the switch, which utilizes the existing DSCP mechanisms to add the packet to the appropriate priority class output queue (8 queues are depicted in the figure). The configurations determine whether D-TAIL is configured for size-blind mode or size-aware mode.

The different underlying transport protocols maintain this information in state differently, depending upon whether they are sender-driven (e.g., DCTCP, pFabric) or receiver-driven (e.g., Homa). We discuss separately how we utilize each type of transport protocol's state to provide the information needed by D-TAIL.

Sender-driven protocols

DCTCP. DCTCP is a TCP-like protocol for data center networks, leveraging Explicit Congestion Notification (ECN) in the network to provide feedback to the senders to control the rate of sending packets. In DCTCP, D-TAIL is implemented on the sender side where the sender stores the flow information and computes the priority of each packet depending on the configuration (size-blind/size-aware) being used. The sender sends a packet as soon as it is ready to send (and the

congestion control mechanism allows it), with the D-TAIL priority level set in the DSCP bits by the sender. As DCTCP does not utilize priority levels, D-TAIL enables strict priority scheduling in the switches. If there is congestion at the priority level the last packet was sent at, the congestion control mechanism will react appropriately and slow down the flow of packets.

pFabric. In this section, we describe the D-TAIL enhancement in pFabric[2], as pFabric is considered to be the optimal protocol for minimizing average FCT. In pFabric, the priority (e.g., SRPT: flow's remaining size) of a packet is encoded using the integer data type by the end-hosts (sender) in the header of every packet. The switches are responsible for identifying the highest priority flow and sending it over the network. Using D-TAIL, the end-hosts (sender) simply perform our priority computation, described in §2.3.1, to calculate the priority for every packet, stored as a float data type, of each flow. Other approaches such as PIAS [22] and pHost[23] are similar to pFabric where senders determine the priorities of packets and set the priority levels provided by either the commodity switches or customized hardware switches for flow prioritization. D-TAIL enhancement mechanism can be implemented in all of these protocols with simple modifications in priority assignments for flows.

Receiver-driven protocols

Homa[1] is a receiver-driven low-latency transport protocol using network priorities. Each receiver (end-host) computes the priorities for all of its incoming data packets in order to approximate the SRPT policy. These priorities are determined based on the flow size distribution with the goal of an equal distribution of bytes across all priority classes. The receiver disseminates the priorities to the senders by piggybacking it on other packets. The senders mark the packets with the priorities obtained from the receiver. For implementing D-TAIL mechanism in Homa, as shown in Figure 2.1, each receiver utilizes D-TAIL's priority computation instead of SRPT and the priority quantization strategy, described in §2.3.2, to determine the priority for every packet. No changes are made at the sender side and in the process of disseminating priorities from the receiver to the sender. Similar to Homa, D-TAIL mechanism can be implemented in any receiver-driven protocol

using flow prioritization.

2.3.4 Bayesian Optimization

Our results in the following sections use values of α that were selected using Bayesian optimization [24]. The approach is general and can utilize any performance objective, in this chapter we focus on 99th-percentile FCT as our performance objective. Bayesian optimization uses Gaussian processes to model an approximation of how α affects 99th-percentile FCT. The optimizer follows an iterative process in which it decides a set of $\alpha \in [0, \infty]$ values to execute for the workload and uses the results to decide the next set of values. Building these relationships between α and the performance metric allows us to efficiently test and quickly find a alpha value for the performance metric for a given workload. §2.4.2 shows the results regarding how the performance improvement increases over the number of iterations and how quickly the optimizer can identify a α value that maximizes improvement.

In this chapter, we focus on the efficacy of D-TAIL using the values of α selected by the Bayesian optimization process. We also show that this process selects "good" αs fairly quickly, and we leave it as future work to demonstrate that α can be adjusted dynamically at run-time based on recent workload histories.

2.4 Simulation Evaluation

The results in this section demonstrate, through packet level simulation over five different types of workloads, that values of α can be quickly identified via Bayesian Optimization that allow D-TAIL to significantly reduce 99th-percentile FCT with minimal impact on overall FCT. The simulations allowed us to explore a variety of workloads upon a larger and more varied set of topologies.

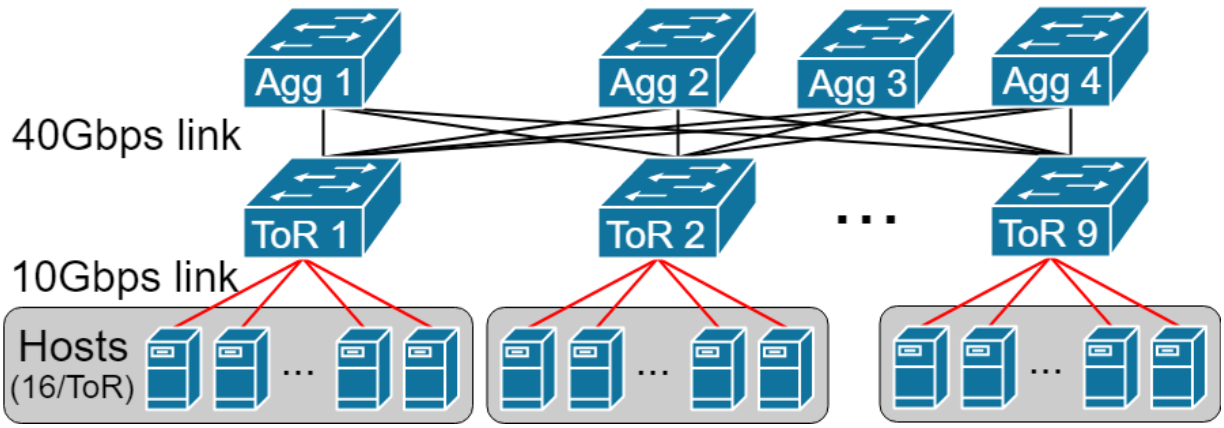


Figure 2.2: 2-tier leaf-spine network topology.

2.4.1 Simulation Setup

We extend existing simulator implementations of the transport protocols to incorporate the D-TAIL mechanism. Specifically, we extended NS2 [25] for pFabric, OMNET++ [26] for Homa, and NS3 [27] for DCTCP.

Topologies

Figure 2.2 shows the 2-tier leaf-spine topology, which is the same topology used for prior evaluations of Homa [1], pFabric [2], pHost [23], and PIAS [22]. It consists of 144 hosts, 9 ToR switches with 16 hosts per ToR switch and 4 aggregate switches. The link capacity of host-ToR links and ToR-aggregation links is 10Gbps and 40Gbps respectively. The link propagation delays are assumed to be 10ns for host-ToR links and 100ns for ToR-aggregation links. The delays are based upon speed-of-light propagation and cable length estimates. The switches use ECMP routing.

Workloads

We used various datacenter workloads, obtained from previous works [28, 29, 9, 1], to evaluate our design. The workloads include Facebook Memcached workload (W1), search application at Google workload (W2), aggregated workload from all applications at Google datacenter (W3),

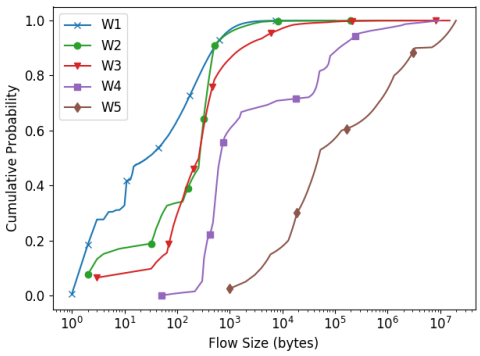


Figure 2.3: *Workload Distributions*. W1 - Facebook Memcached; W2 - Google Search; W3 - Google Mixed; W4 - Facebook Hadoop; W5 - Microsoft Search

Facebook Hadoop workload (W4), and Microsoft Web Search workload (W5) used for DCTCP paper. Figure 2.3 depicts the CDF of the flow sizes for each of the five workloads. While none of the workloads strictly follow the mathematical definition of "heavy-tailed", having a high coefficient of variation (CoV) is often commonly termed as a distribution being "heavy-tailed". Looking at the CDFs, W1 and W2 have relatively compact support, and can thus be classified as "light-tailed" whereas in W3, W4 and W5 the flow sizes show a high range and thus they have a high CoV or are "heavy-tailed".

Workload Generation

In our experiments, unless otherwise stated we pick random source and destination hosts (not belonging to the same rack), with each source generating Poisson arrivals such that the aggregate network load is 80%, similar to Homa[1]. The size of each flow generated is determined via inverse transform sampling on CDFs corresponding to one of the 5 workloads previously discussed.

Configurations for Homa, pFabric and DCTCP

Homa, pFabric. Since we used the original implementations for both of these protocols, we kept the default configurations and simply added our enhancement mechanism on top of them.

DCTCP. We obtained an NS3 implementation of DCTCP from [30], which bases critical functionality on built-in support for TCP RED; this is presently the only such transport protocol in

NS3 that implements ECN marking out of the box. The hosts and the switches utilize 8 priority queues. To meaningfully compare the performance of DCTCP and DCTCP with D-TAIL, we set the DCTCP threshold k for the 8 D-TAIL priority queues to 1/8th the value set for vanilla DCTCP and we do the same for switch queue buffer sizes, thus ensuring the two approaches use similar resources and similar aggregate buffering.

2.4.2 Simulation Results

We now present key results from our simulations. We initially present the trace of how the priority value of a flow changes over time, which provides insight into the D-TAIL priority scheduling mechanism for both size-blind and size-aware. Subsequently, we present transport protocol-specific results showing enhancements by D-TAIL.

Priority value fluctuations for size-blind mode

Figure 2.4 plots the log-base2 of priority values of packets for a single flow in Facebook Memcached workload executed on DCTCP with D-TAIL mechanism for $\alpha = 0.1$ and $\alpha = 1$ on the left y-axis along with the priority class (marked as 'X') of the packet on the right y-axis. The highest priority class is 7, while the lowest is 0. The x-axis represents the packet sequence number i.e. the order in which the packet arrives. This flow was chosen at random from the top 10% longest flows in the workload. This large flow shows multiple instances when the flow is starved and how D-TAIL increases the priority of this flow to attain more service.

The first packet in a flow does not have a priority value and is always assigned the highest priority (omitted in the plot) regardless of the mode D-TAIL operates. For $\alpha = 0.1$, there is an overall increase in the priority value as the scheduling policy is closer to FCFS (where $\alpha = 0$). This is evident as longer the flow is in the system, the later arriving packets (higher packet sequence number) get scheduled in higher priority classes. Even though this flow is not scheduled in the highest priority class 7 ever after its first packet, the log-base2 priority value sharply increases for the last packet. The lack of packets in the highest priority class is simply due to the current

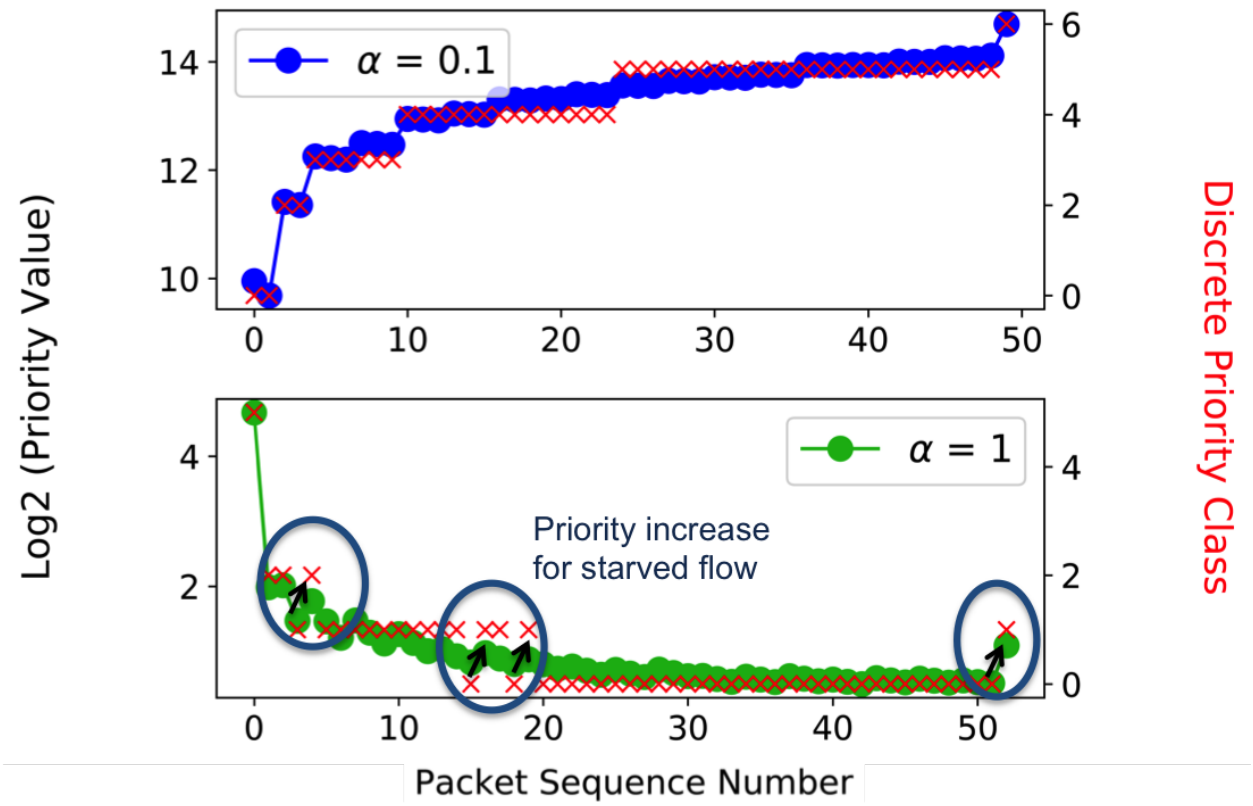


Figure 2.4: Size-blind: Log-base 2 of priority value

selection of priority bin boundaries.

In the bottom subplot, For $\alpha = 1$, the priority values decrease as the flow attains service until it is starved. At starvation, the age of flow increases along with an increase in the priority value of the subsequent packet. There are multiple such instances and one such instance is observed from packets 14,15,16 where packets 14 and 16 are in priority class 1, while packet 15 is in priority class 0. Particularly interesting is the increase in the priority for the last packet, which would have starved without D-TAIL mechanism and suffered a worse FCT.

Priority value fluctuations for size-aware mode

Figure 2.5 plots a similar graph as Figure 2.4 for packets of a single flow in Facebook Memcached workload where $\alpha = 1$ along with the priority class (marked as 'X') of each packet on the right axis. After the first packet (omitted in the plot) is scheduled in the highest priority class 7, the flow has very low priority for initial packets 1 and 2 as bytes remaining is large. In SRPT, this

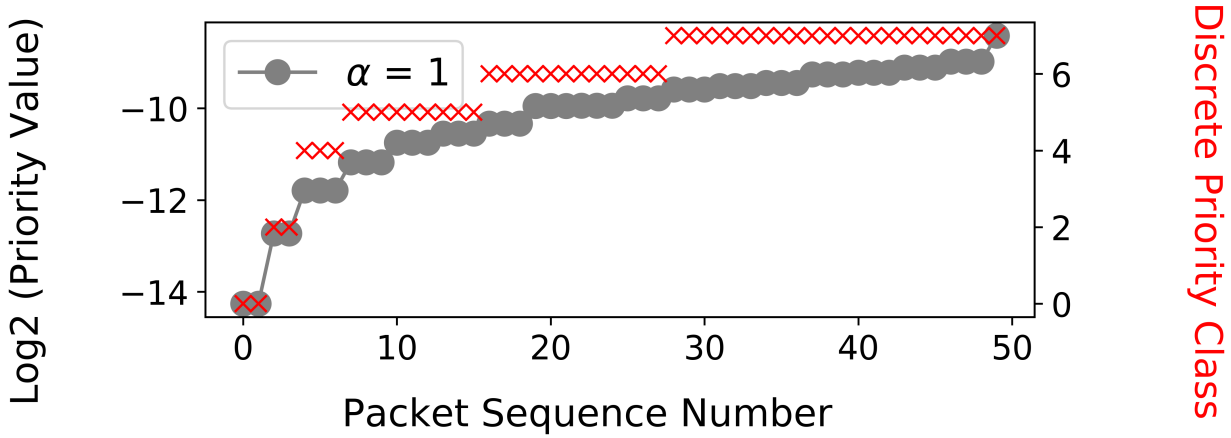


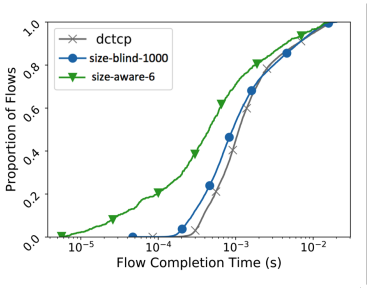
Figure 2.5: Size-aware: Log-base2 of priority value

long flow will be always starved in favor of the short flows. However in size-aware, as the age of flow increases, the priority values increase, thereby scheduling the packets in the higher priority classes. The last packet of the flow is scheduled in the highest priority class 7. This shows how size-aware provides an enhanced SPRT scheduling policy.

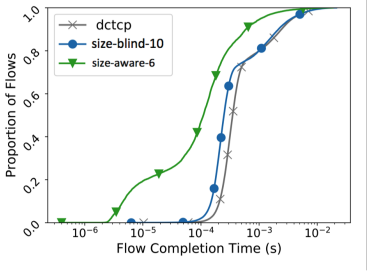
DCTCP Results

We implemented D-TAIL in DCTCP and evaluated both the size-blind and size-aware modes. Figure 2.6 plots the cumulative distribution FCT of all flows for each workload at 80% network load. Flow completion time is plotted on the x-axis in log scale. Each plot includes 3 different curves: one showing vanilla DCTCP while the remaining showing DCTCP enhancements in size-blind and size-aware configurations with α mentioned in the legends. For each workload, flows in size-aware and size-blind modes complete earlier than DCTCP, thereby showing improvements obtained from D-TAIL without compromising the performance of short flows. For instance, in Google Search workload, when half of the flows are completed using DCTCP at $70us$, 90% and 99% flows have been completed for size-blind and size-aware respectively.

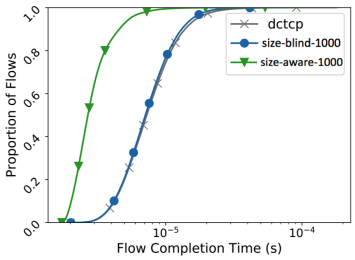
We observed the highest improvement to 99th-percentile FCT across all workloads in DCTCP, as well as improvement in short-FCT, as DCTCP is a more general mechanism that operates in blind mode (flow sizes not required), and hence generally underperforms compared to pFabric and



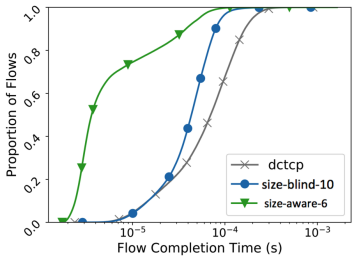
Microsoft Search workload



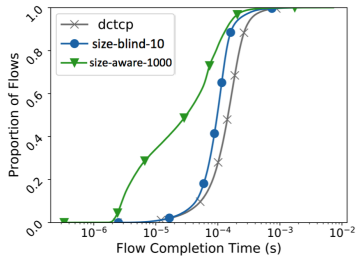
Facebook Hadoop workload



Facebook Memcached workload



Google Search workload



Google Mixed workload

Figure 2.6: DCTCP simulation results: CDF of FCTs.

Homa. As in Homa, to show the enhancement for small flows, we created 10 log-spaced buckets based on the range of flow sizes for each workload, and find the 99th-percentile FCT of all flows falling into each bucket. For example, in Google Search workload, the 99th%-tile FCT is reduced by 59.26% and 74.07% for size-blind and size-aware respectively over DCTCP for the bucket with flow sizes up to 110 bytes. In Facebook Memcached workload, D-TAIL size-blind and size-aware modes provide improvement of 20% and 76% respectively. Similar improvements are obtained for the other workloads and Table 2.1 shows the results for all workloads for the bucket with lowest flow size. Note that the improvement obtained from size-aware mode is always higher than the size-blind mode.

Workload	**Bucket flow size (bytes)**	**99%-tile FCT reduction**	
		size-blind	size-aware
Facebook Mem-cached	350	20%	76%
Google Search	110	59.26%	74.07%
Google Mixed	110	42.19%	68.75%
Facebook Hadoop	1040	35%	41.67%
Microsoft Search	1040	41%	43.33%

Table 2.1: DCTCP 99%-tile FCT percent reduction

pFabric Results

While the improvements of D-TAIL on top of DCTCP are significant, D-TAIL can also enhance highly optimized transport protocols such as pFabric.

Comparisons. Size-aware D-TAIL is compared against the original pFabric implementation, which uses SRPT policy. We also modify pFabric to implement a size-blind mode that applies a Least Attained Service (LAS) policy, since LAS has been shown to minimize average FCT for

workloads where flow-size information is not available [31]. This allows us to look at how D-TAIL can improve a general scheduling approach in both size-aware and size-blind contexts.

Improvement in 99th-percentile FCT. Figure 2.7 plots the percent improvement in pFabric and Homa in the 99th-percentile FCT (x-axis) and mean FCT (y-axis) for each workload using size-aware mode. All workloads show improvement in the 99th-percentile FCT. In this section, we will describe pFabric results, while Homa results are described in §2.4.2. For pFabric, Microsoft Search Workload shows the highest improvement across all workloads at 29%. Similar improvements on a smaller magnitude are seen for mean FCT across all workloads except for Facebook Memcached workload which performs slightly worse at -1.2%. These results highlight that size-aware mode improves the 99th-percentile FCT without significantly degrading the mean FCT.

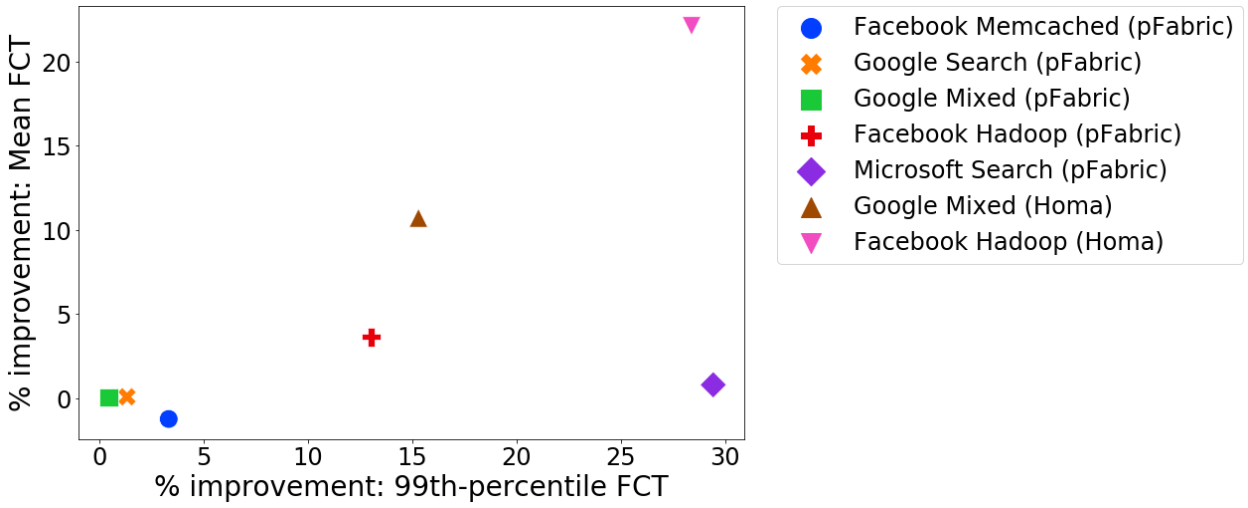
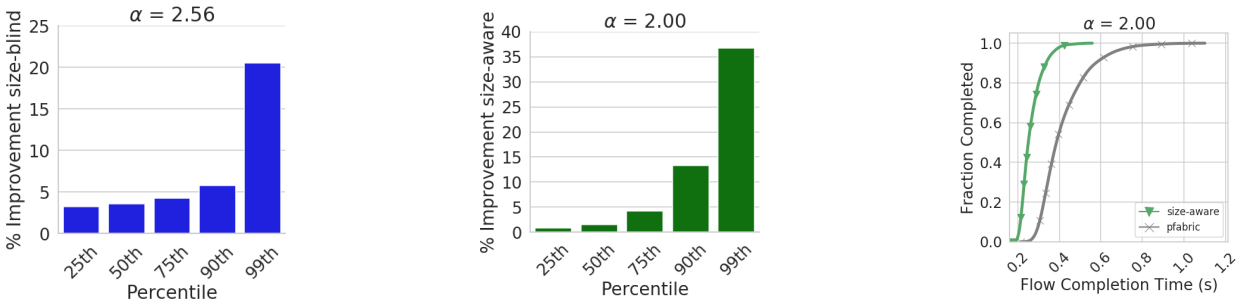
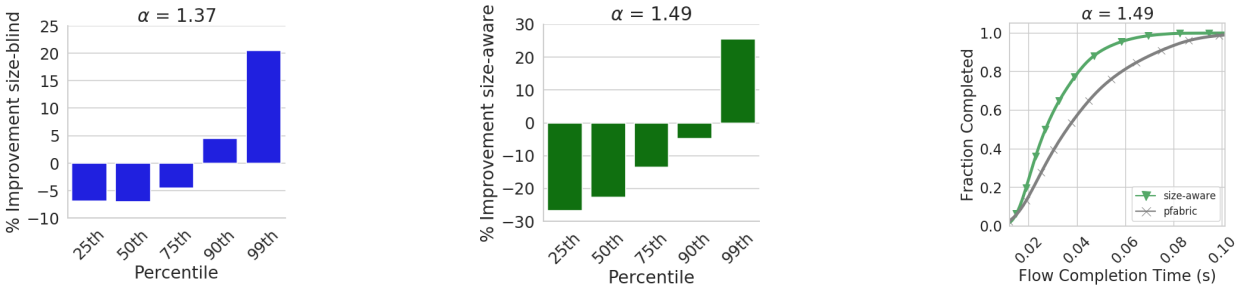


Figure 2.7: pFabric and Homa results: Tradeoff between 99th-percentile FCT and mean FCT

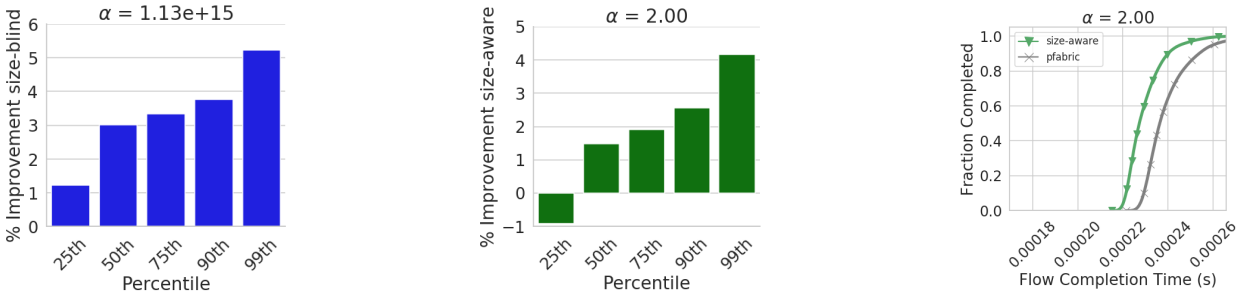
Optimizing for other metrics. Datacenters use 99th-percentile FCT of a workload as a measure of its performance. This metric looks at the FCT of a flow which lies at the 99th-percentile. All the flows after the flow at 99th-percentile, which are potentially starved, are not considered in the metric. Therefore, we also look at an another metric, 99th-percentile mean FCT, that considers these long flows and provides a metric for identifying starvation. The 99th-percentile mean FCT is the mean of the FCT of all flows at the 99th-percentile and above. We run the experiments again with the Bayesian optimizer identifying the α that minimizes the 99th-percentile mean FCT for



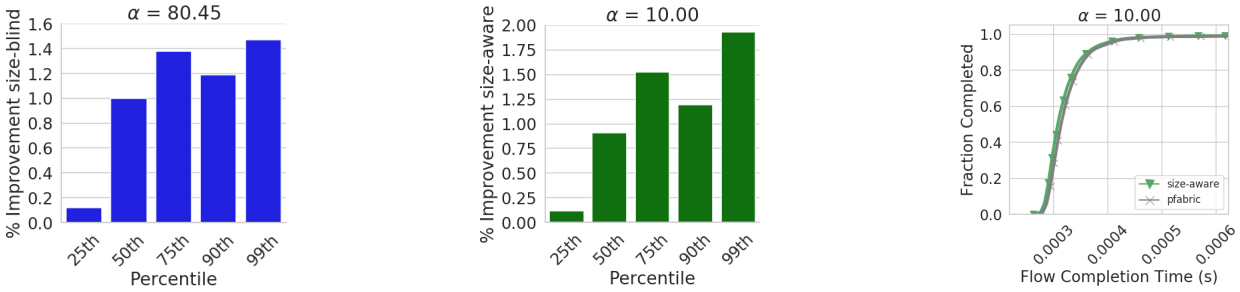
Microsoft Search workload



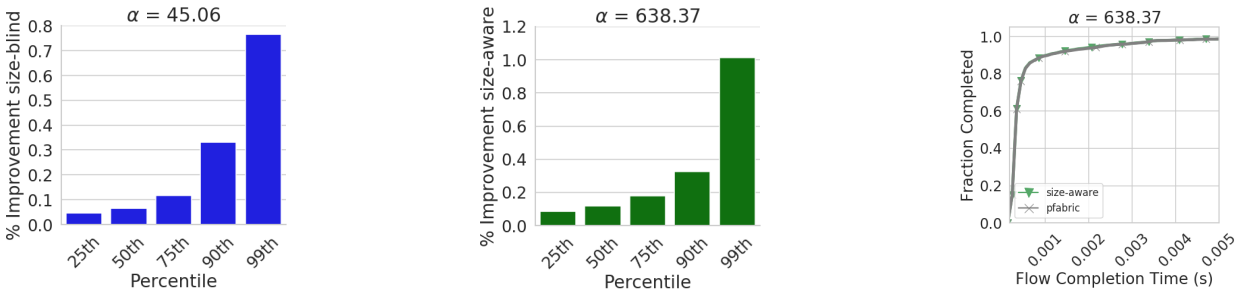
Facebook Hadoop workload



Facebook Memcached workload



Google Search workload



Google Mixed workload

Figure 2.8: pFabric simulation results. Size-blind % enhancement, Size-aware % enhancement and CDF of FCT of top 1% slowest flows, shown from left to right

each workload.

Figure 2.8 plots the percentage improvement using size-blind mode, percentage improvement using size-aware mode, and cumulative distribution of FCT of the 1% of slowest flows for each workload. These improvements are shown for the α value, mentioned on top on each plot, obtained from the Bayesian optimization. The bar plots in Figure 2.8 show the percent improvement at 25th, 50th (median), 75th, 90th and 99th percentile mean FCT for size-blind (left plots) and size-aware (center plots).

Value of α Depends on Workload. The results obtained for Figure 2.8 show that different values of α work better for different workloads, and one size doesn't fit all. Specifically, in the size-blind setting, $\alpha = 45.06$ works best for the Google Mixed workload, whereas $\alpha = 1.37$ works best for Facebook Hadoop workload. Similarly, in the size-aware setting, $\alpha = 638.37$ and $\alpha = 1.49$ provide improvement for Google Mixed workload and Facebook Hadoop workload respectively. With the D-TAIL enhancement, the α can be identified to work well with different kinds of workload distributions, which is a limitation with other approaches which use specific scheduling algorithms. The value of α for Facebook Memcached in size-blind mode is greater than 10^{15}. This high value of α indicates that the priority of a flow is dictated entirely by number of bytes sent, thereby approximating LAS policy.

Magnitude of performance improvements depends on Workload Distribution. From Figure 2.8, D-TAIL improved the 99th-percentile mean FCT for each workload with improvements up to 20% and 37% in size-blind and size-aware respectively for Microsoft Search workload. In Facebook Hadoop and Facebook Memcached workload, this improvement comes at the cost of slowing the short flows, indicated by negative % improvement in the 25th percentiles. For Facebook Memcached, Google Search, and Google Mixed, the percent improvement is less than 5% for the 99th-percentile as 90% of the flow sizes in these workloads are smaller than 1000 bytes, which can be transmitted in a single packet. The improvements in the 99th-percentile are greater than 10% for W4 and W5 as they comprise of a majority of large flows where D-TAIL prioritization is effective and enhances pFabric.

Mitigating starvation for long flows. The CDFs in Figure 2.8 plot the 1% slowest FCTs, which are the longest flows in the workload, for pFabric and size-aware (same value of α that optimizes the 99th-percentile mean FCT). These plots show that the 1% slowest flows complete earlier in size-aware than in pFabric. For instance, in Microsoft Search workload, 99% flows are completed in size-aware while only 50% flows have finished in pFabric at FCT of 0.4 seconds. This indicates that the long flows get prioritized and finish earlier (preventing starvation) in addition to improvements at the key performance metric of 99th percentile FCT.

Finding the α using Bayesian Optimization.

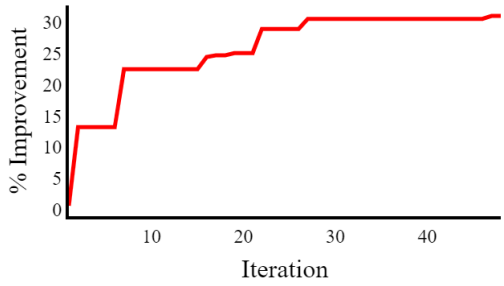


Figure 2.9: Number of iterations for Microsoft Search in pFabric

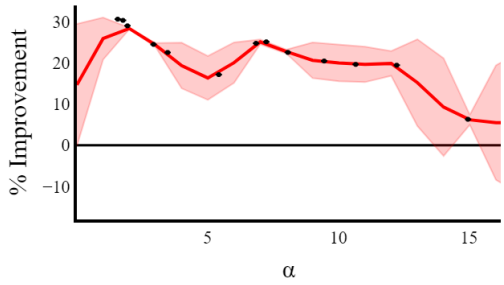


Figure 2.10: Optimizer model of α for Microsoft Search in pFabric

We used Bayesian Optimization to identify the α minimizing workload 99th-percentile FCT. The optimizer runs multiple epochs sequentially, where each epoch consists of up to 4 parallel runs. In our simulations, we ran 24 epochs (96 α values) and found them to be sufficient in identifying the α value that shows improvement for each workload. Figure 2.9 shows the percent

improvement of size-aware over pFabric over the steps of the optimization process. This process is characterized by large early gains in the first 20 iterations and finer gains over the course of the optimization. Figure 2.10 plots the effect of α on % improvement in 99th-percentile FCT. The black dots represent the results from the simulation of the α values selected by the optimizer. The red line is the predicted improvement in the 99th-percentile FCT based on the model, with a shaded red region indicating its corresponding 95%-confidence interval (CI). The figure shows that $\alpha = 2$ gives the highest percent improvement of 30%. The algorithm chooses more points in that region of α to shrink the CI to best exploit expected gain. While running the optimizer for more iterations will lead to tighter CI, size-aware obtained high improvement in fewer iterations.

Homa Results

Similar to performance improvements with D-TAIL on pFabric, D-TAIL on Homa improved 99th-percentile FCT performance compared to the existing Homa implementation. Figure 2.7 shows the % improvement in the mean FCT and 99th-percentile FCT for D-TAIL across workloads. D-TAIL reduced 99th%-tile FCT by 28.39% for Facebook Hadoop workload along with reduction in mean FCT of 22.11% using $\alpha = 0.03$. Similarly, D-TAIL reduced 99th%-tile FCT and mean FCT for Google Mixed workload by 15.30% and 10.71% respectively using $\alpha = 0.0014$. For Facebook Memcached and Google Search workloads, we see minimal improvement of 0.25% in 99th%-tile FCT because 95% of flows are less than 1000 bytes and these flows are transmitted entirely in Homa's mechanism of sending the initial RTTbytes for each flow with pre-allocated priorities.

2.5 Testbed Evaluation

In this section, we describe D-TAIL implementation on the testbed and the evaluations which show the performance improvements obtained for incast, permutation and random workloads.

2.5.1 Implementation

This section presents the implementation of D-TAIL on the testbed of 16 hosts, 2 top-of-the-rack switches and 2 aggregate switches. The testbed, shown in Figure 2.11, is hosted on CloudLab. The link capacity of each link is 10Gbps, achieving a 4:1 oversubscription ratio.

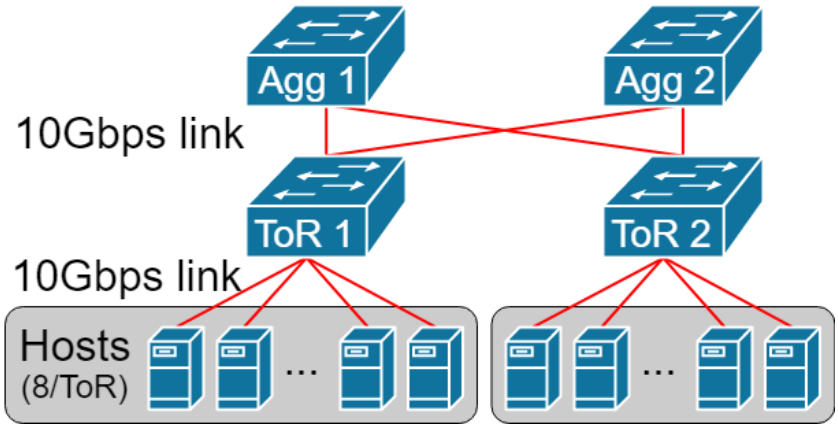


Figure 2.11: CloudLab topology

End-host implementation The hosts on CloudLab, which are identified by the hardware type 'xl170', are 10-core (20 thread) Intel Xeon Broadwell E5 machines with 64 GB of RAM and 480 GB of SSD disk space, running Ubuntu 18.04.1 LTS with 4.15.0-47-generic kernel and DCTCP enabled. We run sender and receiver processes simultaneously on the hosts to track the FCT.

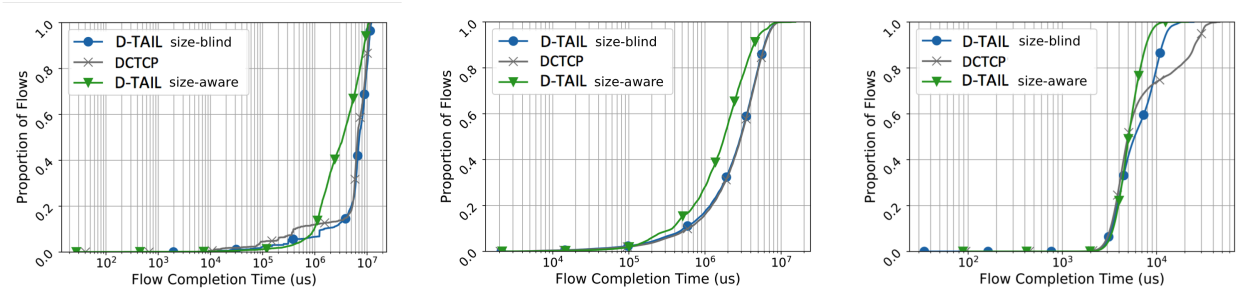
To calculate the priority value for each packet, we use eBPF[32] at the egress interface of the hosts to track the flow size, bytes sent so far and flow age. The eBPF code works as a tc_filter[33] to the interface: it runs in the kernel space, attached to the network stack. The kernel passes each egress packet by calling our eBPF function in which each flow is tracked using the 4-tuple of IP addresses and port numbers in the packet header. For each packet, our eBPF function updates the bytes sent and age of flow, calculates the priority value and writes the tos bits to the packet header, then the packet is returned to the kernel. To speed-up the priority computation and avoid floating point arithmetic, we take the logarithm of the original priority function mentioned in sec. 2.3.1. Then the DSCP bits in the IP header of each packet are marked based on its priority class.

To send a flow, the sender randomly selects a destination from the hosts on the other rack, and

chooses a flow size based on the workload distribution. The connections are closed as soon as the chosen workloads are sent. At the receivers, we keep records of the completion time of each flow, measured by the time from the socket connection was accepted to the receipt of connection closed by the sender.

Switch implementation The switches used in the topology are Dell System S4048-ON [34] with Dell EMC Networking OS version 9.13.0.1. The switches have 8 queues. However, 4 queues are reserved for control traffic, which forced our D-TAIL implementation to utilize only 4 queues. The queues are configured with strict priority scheduling.

2.5.2 Testbed evaluation



Incast workload | Permutation workload | Random workload

Figure 2.12: DCTCP testbed results for incast, permutation and random traffic patterns

In this section, we present the results of the evaluation on the CloudLab testbed. The topology is shown in Figure 2.11. As previously noted, the switches available on cloudlab reserve 4 priority classes for control traffic, so we are limited to using 4 priority classes instead of 8 as in the simulations. In addition, only the switches utilize the priority queues unlike simulations where both the hosts and switches were configured with priority queues.

The results, however, continue to follow a similar pattern of D-TAIL enhancing performance significantly of DCTCP, both for FCT as well as controlling tail latency. We tested D-TAIL in a variety of traffic characteristics such as incast, permutation and random. These traffic patterns have been used for evaluation in [8, 10]. In incast, 15 hosts send 1350KB flows to one receiver host. In permutation, each host sends to and receives flows from only a single host. In random,

the source and destination hosts are chosen at random. Figure 2.12 plots the CDF of FCT of flows for the best α for the three workloads. For incast and permutation workload, size-aware ($\alpha = 4$) enhanced DCTCP, while size-blind had similar performance as DCTCP. For random workload, both size-blind ($\alpha = 2$) and size-aware ($\alpha = 2$) enhanced DCTCP.

2.6 Discussion and Future Work

In this section, we discuss and propose future work in D-TAIL that will be promising and further improve the datacenter network performance.

Partial availability of flow size information. D-TAIL works in either the size-blind mode (flow-size information is not available) or the size-aware mode (flow-size information is available). Given the two extremes in terms of flow-size information available, naturally a question arises whether D-TAIL can have a mode relying on partial availability of flow-size information. The partial availability of flow-size information can be obtained in two ways. Firstly, as shown in [15], flow-size information can be obtained for certain workloads with a accuracy greater than 80%. In this case, D-TAIL's priority computation can utilize the predicted flow-size information to compute the priority class for the flow. In future work, it will be interesting to explore if the partial information can reduce the tail latency of the transport protocols. Also, identifying the percentage of prediction accuracy required would be beneficial to classify the mode of usage in D-TAIL for a workload. Secondly, a datacenter can have a diverse mix of applications and the application developers for a subset of applications can provide the flow-size information to the transport protocol. It is worth investigating in the future if using a hybrid of size-blind mode and size-aware mode, i.e. using size-blind for flows without flow-size information and using size-aware for flows with flow-size information, can improve the overall performance of the datacenter networks.

Tail-Latency as a function of α. In D-TAIL, the tunable parameter α is powerful because it can be fine-tuned to achieve better performance for a variety of workloads. However, finding the value of α requires the use of Bayesian optimization. In the optimization, multiple iterations

of the experiments are required to identify an α that improves performance. While all of our experiments have shown that 25 iterations are sufficient for the analyzed workloads, it is entirely plausible that significantly more iterations can be required for other workloads. We believe that understanding the impact of α on tail-latency would be beneficial. One opportunity to explore is finding tail-latency as a function of α dependent on factors including workload characteristics such as flow-size distribution and flow arrival rate, and network characteristics such as topology, link-speeds, and number of priority classes in switches. Having this function can further enable datacenter operators to easily figure out the value of α for their workloads.

Optimally selecting the bin boundaries in Priority Quantization Priority Quantization is dependent on the number of priority classes supported by the switches. Typically, commodity switches support 4 or 8 priority classes. In priority quantization, we aimed to distribute equal number of bytes across the available priority classes. This distribution required splitting the infinite range of priority values into 4 to 8 bins depending on the switches. In our simulations assuming 8 bins, we had to identify 7 priority values that would serve as the bin boundaries. We observed that the tail latency for a workload was extremely sensitive to the selection of bin boundaries. Obtaining the bin boundaries that reduced tail-latencies was a challenging task due to the time factor of age of flow in the priority calculation. The age of flow makes it difficult to get an even distribution of bytes across the priority classes. One possible strategy worth exploring in the future work is using Bayesian Optimization for selecting the bin boundaries. This approach adds complexity due to the need for fine-tuning multiple parameters. However, it would immensely simplify the protocol deployment for the datacenter operators.

2.7 Related Work

§2.2.1 describes how the adaptable priority scheme, PBS [17], is applied within D-TAIL to dynamically adjust flow priorities within a network as a function of their age and bytes sent. Our current work makes two contributions. First, PBS was proposed as a general purpose scheduler in any $M/G/1$, and D-TAIL is the first to demonstrate the applicability of PBS within a networking

context. Second, PBS was designed as a configurable blind scheduler, while our work here extends the mechanism in size-aware settings, giving better performance to blind PBS by reducing the flow completion times for 99th-percentile.

A large body of work exists with the intention of providing low latency for short flows and high throughput for long flows in datacenter networks [10, 35, 1, 2, 36, 23, 37]. While the goal is for the most part identical, the underlying differences are with respect to 1) whether flow sizes are known a priori (i.e., use non-blind scheduling), 2) the importance of maintaining high throughput for larger flows, and 3) to what degree the underlying parts that make up the data center (end-systems, switches, and transport protocols) require modification or specialization. With respect to these differences, D-TAIL can 1) operate in-the-blind (without knowing flow sizes) but, as demonstrated in Sec. 2.3.1, utilize flow size information when available, 2) increase its tunable parameter, α, to increase the importance of maintaining high throughput for larger flows, and 3) has a minimalist approach to modifications required for its implementation (i.e., priority classes at switches and a priority marking mechanism at the sender end-systems), enhancing performance of existing transport protocols.

The approach most closely related to ours is [37], which operates in-the-blind and, without knowledge of flow size assigns a flow, assigns a flow to a decreased priority class as the length of bytes transmitted increases. Deciding the size at which a flow transitions to the next priority class is a challenging problem that must be precisely tuned, and is heavily dependent on the statistics of the workload. While D-TAIL also requires deciding the priority bin boundaries, this problem is alleviated in D-TAIL due to the tunable parameter. The tunable parameter allows us to iterate and identify the splitting of bytes across priority classes; thereby, providing an additional and simpler control mechanism for fine-tuning workload performance. Furthermore, since the sojourn time of a flow is not taken into consideration, large flows can become starved indefinitely under heavy network loads, whereas D-TAIL will increase priority of a large flow when the ratio of bytes sent to sojourn time is too small.

To the best of our knowledge, all remaining approaches utilize non-blind scheduling and as-

sume individual flow sizes are known a priori [1, 2]. While these approaches provide lower 99th-percentile flow completion times, they lack flexibility and cannot be utilized for workloads where knowing or predicting the flow sizes a priori is difficult. [15] presents an analysis of workloads with unpredictable flow sizes. For such workloads, D-TAIL size-blind mode can be deployed.

A different approach to non-blind scheduling is to require a pre-authorized scheduling of flow packets (e.g., [23, 36, 35]) and therefore rely heavily on knowledge about the internal configuration of the switching fabric, and must take extra steps to remain fault tolerant.

Rather than provide explicit schedules, some mechanisms require pacing the rate at which senders inject traffic to ensure the various flows conform to the capabilities of the internal switches. For instance, [38] assigns flows to different priorities at the application layer, and sender-side rate limiters ensure controlled entry of traffic into the network. This approach requires analyzing traffic generated by applications, and manually assigning priorities per-application, which places additional burden on datacenter operators. Another paced approach is presented in [39], where the underlying transport protocol is DCTCP, but where switches must also perform phantom queueing. This approach does not utilize priority queues in the switches, which D-TAIL takes advantage of to reduce flow completion times for a variety of workloads.

Work that requires substantial modification to internal switches and to end-system design [2, 40, 1, 8] have demonstrated substantial latency improvement in small flow delivery latencies, and generally claim to maintain sufficient throughput for large flows, but cannot provide any stringent guarantees and conditions can arise in which starvation of large flows can occur.

Chapter 3: Exploiting content similarity to address cold start in container deployments

Container deployments is an important process for the cloud providers. An efficient deployment mechanism can lead to better utilization of resources for the cloud provider and high performance for the users. On the contrary, an inefficient deployment mechanism will create bottlenecks and degrade the users performance, potentially forcing the user to walk away from the cloud provider services. Cold start is one of the critical problems affecting the existing container deployment designs [41, 42, 43]. Cold start is the latency induced in the process of creating a container, downloading and booting the container on a worker node for execution. In this chapter, we propose a container deployment design consisting of peer-to-peer network and virtual file system with content-addressable storage to tackle the cold start problem by exploiting the content similarity within containers.

3.1 Introduction

CISCO[44] has forecasted that 94 percent of workloads and compute instances will be processed by cloud data centers and the annual global data center IP traffic will reach 20.6 ZB by the end of 2021 [45]. This begs the question: *Is the current cloud system capable of efficiently utilizing compute, storage and network resources for the increased workload?*

Serverless computing has emerged as a win-win technology benefiting both cloud providers (CP) by providing fine-grained control and the users by allowing quick application deployments, version control, and attractive pricing [46, 47, 48, 49, 50, 51, 52, 41, 6, 53, 54, 55, 56, 57]. In serverless, the CP maintains generic worker nodes to execute the functions uploaded by the users from a centralized *storage bucket* (e.g., S3[58]) that holds code to be executed when corresponding

jobs arrive. For function execution, the CP builds a container with the source code, downloads the container on a worker node from centralized storage bucket and executes it. This process induces a latency while downloading and initializing the container, referred to as a "cold start", whose magnitude is dependent on a set of application-specific factors [41, 42, 43]. The cold start problem is exacerbated with scaling as each new function execution incurs the cold start latency. To reduce the impact of cold start for subsequent executions, multiple instances of the worker node are retained for tens of minutes [43], thereby reducing compute availability.

The cold start problem arises due to the existing container deployment design, which retrieves each new instance of a container from the storage bucket. Our proposed system design addresses the problem and consists of 4 modules: peer-to-peer network (p2p), virtual file system with content-addressable storage (CAS), partial delivery execution and TCP splitting. The user uploads the source code to a CAS bucket that has hash based references to file blocks using a MerkleDAG [59]. Each worker node runs a p2p client and a CAS-based virtual file system. To execute the containerized source code, the worker node requests file blocks from the virtual file system, which serves the blocks if already present at the node; otherwise, it fetches the requested file blocks from the p2p network from other nodes and/or the CAS bucket. This enables partial delivery execution[60] (starting execution before all blocks are delivered) of the container, with the added benefit of live migration [61] of the container from one worker node to another. Live migration requires availability of the application's IP address on the destination machine, which we ensure via TCP splitting at the master node as shown in figure 3.1. Our system can be incorporated in existing container orchestration platforms such as Kubernetes [62].

3.2 Key Ideas

The key ideas are based on analysis of the characteristics of the different ways the applications are deployed, and motivating the need for hash-based reference to file blocks.

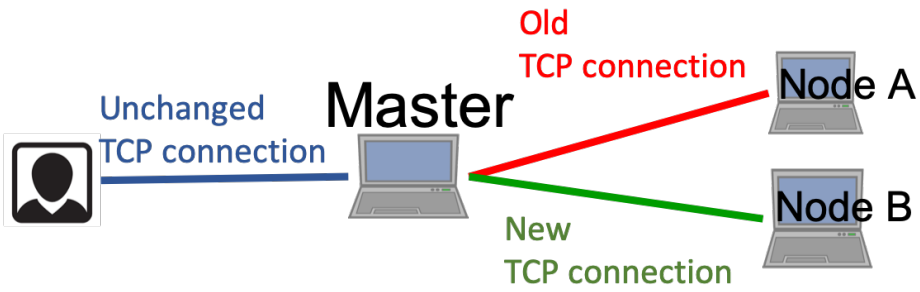


Figure 3.1: TCP splitting

3.2.1 Characteristics of deploying an application

Scaling [63], versioning [64], and live migration [61] represent the three ways in which an application is deployed on the worker nodes. In scaling, multiple identical application instances are replicated across nodes to meet either the current demand or predicted demand. In versioning, the application is updated, requiring different versions to be supported on the cloud. The updated application will likely have significant overlaps of data with previous versions. In live migration, an application is moved from one worker node to another, which requires saving current state (checkpoint) of the container, transferring and executing the checkpoint on a different worker without disrupting the user experience. For all the three modes, our proposed system can reduce the latency by exploiting the presence of identical blocks. Figure 3.2 shows the percentage of identical $256KB$ blocks of 15 checkpoints of an open source e-commerce web application, PrestaShop [65], taken after every minute. As shown in figure, the first checkpoint had 53% blocks identical to the original container checkpoint. Even in later checkpoints there still remained a 24% overlap with the original configuration of the application.

3.2.2 Block similarity in source code across applications

Applications tend to commonly use popular libraries for a variety of tasks, such as image processing, data analytics, machine learning, etc. Based on scraping 876K Python projects from GitHub, [41] identified 20 most common PyPI[66] packages used as GitHub project dependencies and measured the initialization time for these packages with 12.8 seconds as the highest time

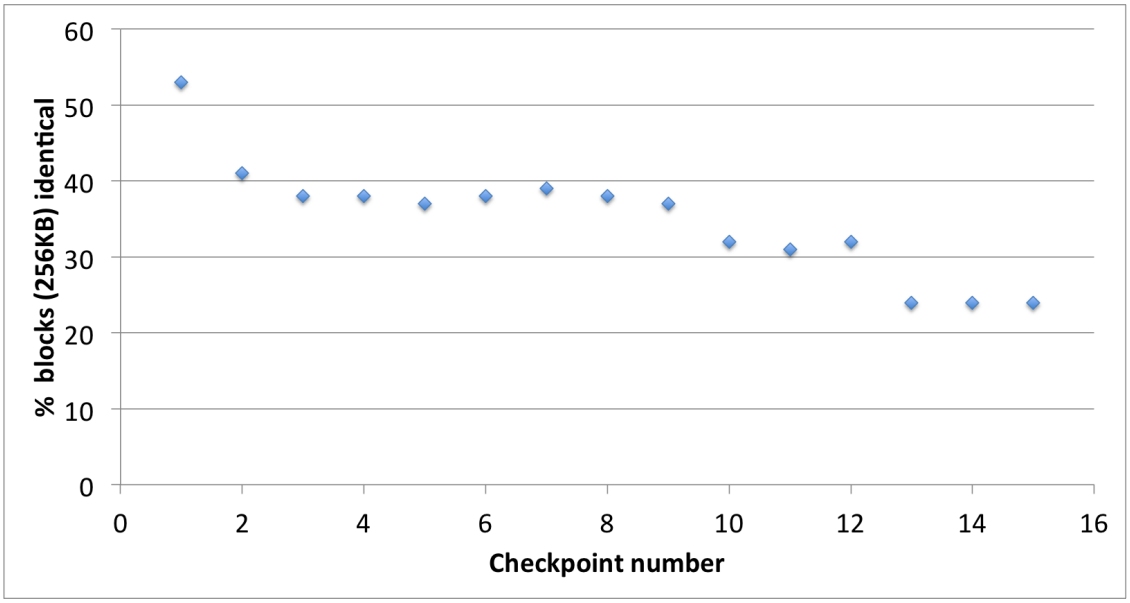


Figure 3.2: Blocks similarity

for pandas[67]. This time can be reduced by our proposed system as the blocks for the popular libraries may already exist on the worker node and/or can be fetched from multiple worker nodes by leveraging the p2p network.

3.3 Implementation and Evaluation

Our system utilizes modified IPFS [68], a p2p, content addressed file system. The IPNS [68] mount-point provides the virtual FUSE-based custom file system enabling container executions and live migrations. The evaluation below shows twofold benefits: minimizing average completion time of container checkpoint transfers and reduction in the cold start latency.

3.3.1 Multiple Shuffles

Our testbed consists of 8 t2.micro[69] nodes on AWS[70], four in Oregon region and the remaining in North Virginia region. Each node has 2 container images (roughly 200MB each). In shuffle, each node acquires 2 container images chosen at random from nodes in the other region. We perform multiple shuffles capturing scenarios in big data distributed processing frameworks such as Apache Spark [71]. Figure 3.3 shows an improvement of 33.3% and 59.3% in average

transfer completion time over rsync[72] transfers for the first shuffle and sixth shuffle respectively.

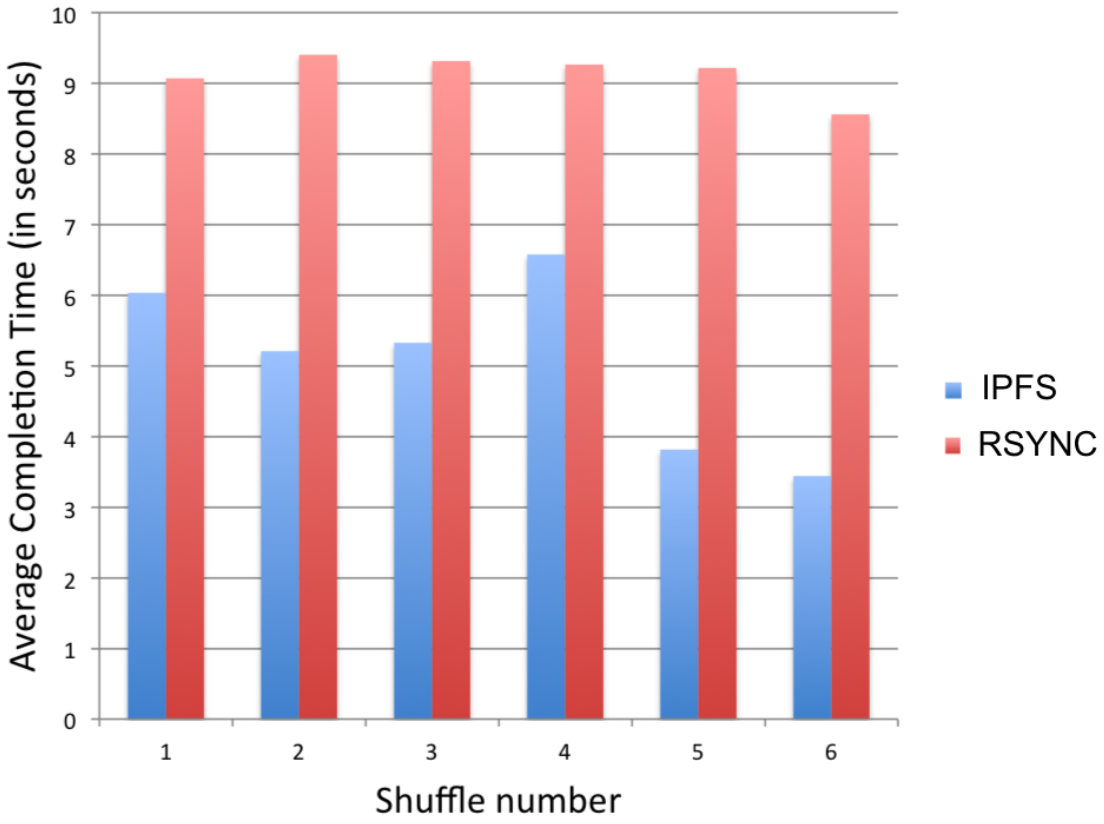


Figure 3.3: Multiple shuffles

3.3.2 Live Migration

We developed a simple web application which simply kept a count of number of HTTP requests and created a checkpoint whose size was 7MB. We performed live migration of the container using our system and found a 37.9% reduction in boot time over rsync on the same testbed as in sec 3.3.1 with 2 nodes containing copies of the checkpoint.

3.4 Related Work

[41] proposed a specialized container system, eliminating kernel bottlenecks, that is optimized for serverless workloads. Other solutions to cold start problem include increasing the memory

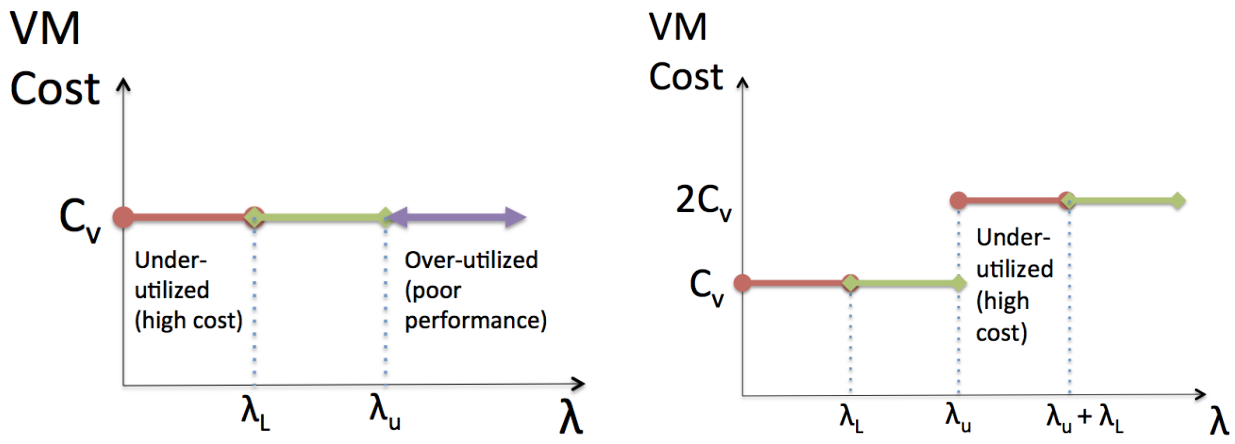
allocation, choosing a faster runtime, keeping shared data in memory, shrinking source code package size, monitoring performance and logging relevant indicators, using time-series forecasting, and keeping a pool of pre-warmed functions [73]. However these solutions not only put an additional burden on the developers by increasing monetary costs, reducing implementation flexibility, requiring significant optimization, thereby hindering quick deployment and fast growth, but also inefficiently utilize the compute and storage resources of the cloud provider.

Chapter 4: Optimal Pricing for Serverless Computing

Cloud computing is a disruptive technology that over the past two decades has dramatically changed how enterprises compute and how computation is monetized. Cloud computing platforms offer a plethora of computing paradigms such as Virtual Machines, Containers, and Serverless Computing along with their pricing models. The enterprises can utilize either one of these services or any combination of them. In this chapter, we provide a smart enterprise user model, which aims to meet the desired performance objectives, minimize monetary costs, and provides the workload breakdown for each of the services. Assuming this user model, we also provide a pricing strategy for the cloud providers to maximize their profits.

4.1 Introduction

The classic computing paradigm is the rental of virtual machines (VMs) for prolonged periods of time: the price depends on the VM resources (e.g., CPU, memory) and contract duration (e.g., month, year).



(a) Single VM (b) Multiple VMs

Figure 4.1: Cost analysis for VMs

This VM rental paradigm can be inefficient for both the customer and provider since the reservation of resources is for a long duration. This inefficiency is explained in Figure 4.1: The left plot shows the cost as a function of the arrival load (e.g., jobs per second). Note that as the VM rental cost is fixed (C_v), the cost does not vary with the load when a single VM is rented (Figure 4.1(a)). There are two noteworthy issues: 1) when the load is too low, the VM is underutilized. Thus, the customer ends up paying a higher price for its effective computation; 2) when the load is too high, the VM is overloaded, and performance degrades to inadequate levels (e.g., very long response times). The second problem can be addressed by renting a second VM, as illustrated in Figure 4.1(b). However, the first problem persists. Note that if the first VM is overloaded by a small amount, renting two VMs can still leave the VMs underutilized.

A more recent paradigm is the rental of VMs under a contract that charges only for the times during which the VM is on. While this paradigm permits both the customer and provider to utilize resources with greater efficiency, there is still an overhead incurred, namely, time required to hibernate the VM, time to restore a hibernated VM, and user responsibility for turning the VMs on and off. In this chapter, we focus on the more classic VM rental model, where the user pays a fixed rate independent of the state of the VM.

Even more recently, the Function-as-a-Service (FaaS) paradigm, also known as **Serverless Computing (SC)**, has emerged, in which users install functions in the cloud that are executed by their remote applications. The cost of a function call depends on its running time and resources consumed such as memory. From the provider perspective, since SC operates atop a container infrastructure as opposed to a more generalized VM infrastructure, the underlying hardware can be utilized more efficiently as it accommodates more containers than general purpose VMs [74]. This allows providers to increase profit by driving user loads onto SC, and to raise the respective price for unit time processing in SC.

However, SC can only handle a certain class of workload (different cloud providers offer different APIs and languages to instantiate the functions) and VMs will continue to be offered by cloud provider. The price of renting the VMs is then decided by market competition between the

cloud providers, and so the providers cannot arbitrarily increase the price of SC (or offer only SC) as the rational user always has the option to switch to (the cheaper) VM rental.

In this chapter, we explore the question of how a provider can relatively price SC, in order to maximize profit when facing rational users that minimize their costs (under some performance constraint). We combine simple cost models grounded in existing cost profiles offered by real cloud providers and traditional queueing theory and game theory to identify optimal strategies for both the user and provider. In particular, we determine how a user should purchase cloud services to minimize its cost and serve its workload under a given performance constraint. Moreover, we determine how the provider should set prices for cloud services in order to maximize profits. More specifically, we make the following contributions:

- We leverage simple queueing models for VMs and SC to characterize their performance under some given load (see §4.2).
- We analyze the model and determine the optimal allocation between VMs and SC as a function of model parameters, such as prices for VMs and SC set by the provider, service capacity of VMs and containers, and user performance constraint (see §4.2.1).
- Three possible operational regimes emerge under minimum user cost: use SC only, combine SC and VM, and use VM only.
- We propose a simple cost model for the provider to offer VM and SC, where its revenue is identical to the user's cost. Under this consideration, we formulate a game between the provider and the user, while for former sets a price and the latter decides on VM and SC (see §4.3 and §4.3.1).
- We identify the analytical expression for a Nash equilibrium of the game, using a strategy where the user minimizes its cost (under a given price) and the provider maximizes its profit (see §4.3.2).

4.2 Cloud Services Model: User perspective

Consider cloud computing services where a client (user) may utilize both serverless computing (SC) and virtual machines (VMs). We apply the traditional cost model of such services where VMs are rented at a fixed price per unit time (denoted by α_v), irrespective of the loads placed on them and the running times of jobs they process. In contrast, SC also has a fixed unit time price (denoted by α_s). However, the cost of SC is proportional to sum of running times (duration) of jobs processed by the service.

Each rented VM is modeled as a single server system with a fixed capacity. However, due to variability in job sizes, we assume the time required to service a job can follow any distribution with average $1/\mu_v$. Note that μ_v can be interpreted as the average job service rate and is a parameter of the VM.

SC is modeled as an infinite server queue with fixed capacity, as the cloud provider dynamically allocates resources to service each job within a given performance profile (e.g., provided enough memory and CPU to process at a given rate). More specifically, the provider dynamically allocates containers (a lightweight VM at OS-level) to service a job arriving to its SC platform [74]. Although a container is given a fixed processing capacity (and memory), the variability in job sizes leads to an arbitrary distribution for the job service time. We assume that the average job service time in SC is $1/\mu_s$. Again, μ_s can be interpreted as the average job service rate and is a parameter of the SC platform.

Consider a client that uses cloud services to process a continuous arrival of jobs, with an average rate denoted by λ (e.g., $\lambda = 10$ jobs per minute), and would like to minimize its cost subject to some performance constraint, such as mean response time of jobs. In particular, we assume the client can leverage multiple VMs and SC simultaneously, splitting its load across the contracted services. We assume that the arriving jobs are mutually independent, i.e. the execution of a job does not depend on the execution of any other job[1].

[1]This assumption can be relaxed considerably to assuming only that two simultaneously running jobs are not dependent on one another.

We assume client load arrives according to a Poisson process with rate λ and is split randomly[2] among the rented resources, such that a thinned Poisson process arrives at VMs and SC. This permits us to model each VM as an M/G/1 queue, while SC as an M/G/∞ queue.

Recalling that the client has an explicit performance constraint when using cloud services, we assume this constraint is an upper bound on the average response time of jobs sent to the cloud for processing. For the SC model, this constraint translates directly to the average service time parameter, $1/\mu_s$. For the VM model, we employ a utilization constraint: every rented VM must have a utilization that is smaller than ρ_t, a parameter set by the client. In this form, using the Pollaczek-Khinchine formula [75], the response time constraint applied to an M/G/1 queue maps to a utilization constraint given the mean and second moment of the workload distribution. Moreover, several other performance constraints, such as the 95^{th} response time, can also be mapped to some maximum utilization in this model.

Let $s(\lambda) \leq \lambda$ denote the rate of load the client assigns to SC. Recall that in SC the client pays as long as the job is being processed. Thus, the average cost of processing this load is given by the average number of busy servers in the M/G/∞ queue, which is given by $s(\lambda)/\mu_s$, times the unit time cost, α_s.

Let $v(\lambda)$ denote the number of VMs rented by the client when its load's rate is λ. Due to the performance constraints, each VM can accept a maximum load of $\rho_t\mu_v$, and thus, the maximum load the rented VMs can jointly accept is $v(\lambda)\rho_t\mu_v$ at a cost of $v(\lambda)\alpha_v$, since the user pays per rented VM.

For a given load λ, the client chooses $s(\lambda)$ (how much load to send to SC) and $v(\lambda)$ (how many VMs to rent) to minimize cost:

$$c(\lambda) = \alpha_s s(\lambda)/\mu_s + \alpha_v v(\lambda) \tag{4.1}$$

subject to $\lambda - s(\lambda) \leq v(\lambda)\rho_t\mu_v$, which is the performance constraint for the rented VMs. Note that

[2]While this is a simple stateless load balancing policy, using more sophisticated load balancing mechanisms does not qualitatively change the nature of our results, it simply changes the threshold ρ_t used in our analysis

Parameter or variable	Description
λ	total user load
$v(\lambda)$	number of rented VMs
$s(\lambda)$	load diverted to serverless
α_v	unit VM cost
α_s	unit time cost of serverless
μ_s	service capacity for serverless
μ_v	service capacity of a VM
ρ_t	utilization constraint for a VM
L_s	load at which cost of serverless equals cost of VM
L_v	maximum load accepted by a VM

Table 4.1: Model parameters and variables.

we assume that $1/\mu_s$ meets the average response time constraint for the client, since otherwise SC would never be considered. Table 4.1 summarizes the model parameters.

4.2.1 Analysis

We now investigate the tradeoffs and economic benefits in renting VMs and using SC. Intuitively, SC can be leveraged to reduce customer costs, especially for low loads or for loads just over the performance constraint.

Lets start by noting that the cost of serverless is linear in its load whereas the cost of VM is independent of its load. For small enough loads, serverless will always be cheaper. However, since this cost increases linearly with the load, it will eventually surpass the fixed cost of renting a VM. Let $L_s = \alpha_v \mu_s / \alpha_s$ denote the load at this crossover. When user load $\lambda < L_s$, the user should only use SC.

When $\lambda > L_s$, renting a VM is cheaper and the VM can handle loads $\lambda > L_s$ until the performance constraint is no longer met. Let $L_v = \rho_t \mu_v$ denote the load that a single VM can accept under the capacity constraint. Thus, for $\lambda \in [L_s, L_v]$, renting a VM is cheaper and can satisfy the performance constraint without invoking SC.

When $\lambda > L_v$, the user must either rent another VM or divert part of the load to serverless.

Again, since serverless has a linear cost structure, it is cheaper to shed small overloads to SC for a small enough excess load. When the excessive load is sufficiently large, it will again be cheaper to rent a (second) VM, and SC will not be used.

This process repeats itself as λ increases, and the optimal allocation can be computed as follows. For a load λ, we need a total of $\lfloor \lambda / L_v \rfloor$ VMs since each VM can handle a load of L_v within the performance constraint determined by the user. Note the number of VMs is an integer. Since each VM can handle a load of L_v, the total load handled by the VMs is simply $\lfloor \lambda / L_v \rfloor L_v$. Given this value, the excess load is given by $\lambda - \lfloor \lambda / L_v \rfloor L_v$. Finally, if this excess load is smaller than L_s, we send it to serverless. Otherwise, we rent an additional VM to process this excess load, as it will be cheaper.

From the above calculations, in general we have the following optimal result for renting VMs:

$$v(\lambda) = \begin{cases} 0, & \text{if } L_v \leq L_s \\ \left\lfloor \frac{\lambda}{L_v} \right\rfloor, & \text{if } \lambda - \lfloor \lambda / L_v \rfloor L_v \leq L_s \\ 1 + \left\lfloor \frac{\lambda}{L_v} \right\rfloor, & \text{otherwise} \end{cases} \tag{4.2}$$

Moreover, we can determine the load diverted to serverless in the optimal allocation, which is given by

$$s(\lambda) = \begin{cases} \lambda, & \text{if } L_v \leq L_s \\ \lambda - \left\lfloor \frac{\lambda}{L_v} \right\rfloor L_v, & \text{if } \lambda - \lfloor \lambda / L_v \rfloor L_v \leq L_s \\ 0, & \text{otherwise} \end{cases} \tag{4.3}$$

Thus, equations (4.2) and (4.3) give the solution that minimizes the cost posed in equation (4.1).

Last, note that if SC is much cheaper than VMs, then renting VM may never be more cost effective. This occurs when $L_v \leq L_s$, indicating that when the load at which the cost of serverless equals that of a single VM is larger than what a VM can handle, then the VM is just not cost effective. In this scenario, we have that $s(\lambda) = \lambda$ and $v(\lambda) = 0$, as indicated in the above equations.

4.2.2 Numerical evaluations

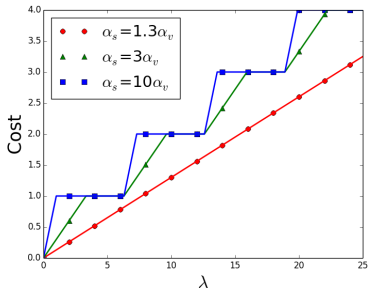


Figure 4.2: Optimal customer cost with $\mu_s = 10$, $\mu_v = 7$, $\alpha_v = 1$.

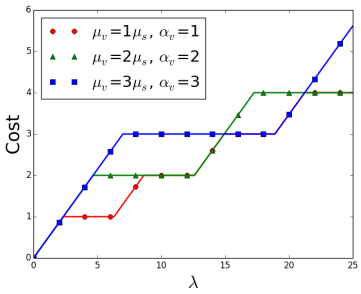


Figure 4.3: Optimal customer cost with $\mu_s = 7$, $\alpha_s = 3$.

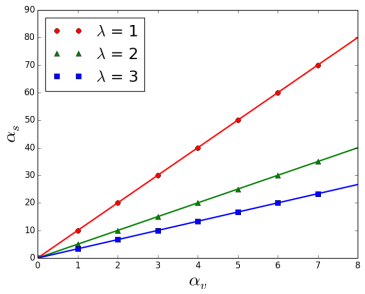


Figure 4.4: Unit time price regions ($\mu_s = 10$, $\mu_v = 5$).

Figure 4.2 plots the optimal cost as function of the load λ for different ratios of α_s to α_v. The curves increase linearly with respect to λ when SC is used and flattens when an additional VM is rented. Note that the cost reduction when using SC strongly depend on the relative prices. If SC is relatively expensive ($\alpha_s = 10\alpha_v$), then the cost savings are marginal. On the contrary, if SC price is moderate, cost reduction is significant for a larger range of loads ($\alpha_s = 3\alpha_v$). Last, if prices are comparable ($\alpha_s = 1.3\alpha_v$), then SC always yields a lower cost.

Another important consideration is the relationship between VM price and capacity, as cloud providers offer a wide range of VM capacities at different prices. Figure 4.3 shows the optimal user cost for different price/capacity ratios. Note that as VM capacity increases (and thus its price), the cost reduction yielded by SC also increases. This occurs because L_s increases with the price of the VM. However, since the VM has more capacity, it also handles more load, inducing a larger L_v.

Finally, we investigate the price regions for optimal allocation of VMs and serverless. Note that given the model parameters (including λ) and a given pair prices α_s and α_v, either VM rental is cheaper, or serverless is cheaper (or the same). This induces regions in $\alpha_s \times \alpha_v$ plane where VM or serverless is more cost effective. Figure 4.4 illustrates these regions for different values for λ. Note that each curve (straight line) corresponds to the case where the costs are equal. Below and above the line correspond to the cases where SC and VM rental is cheaper, respectively. Last,

Parameter	Description
β_s	unit time cost to the cloud provider to offer a container used in SC
β_v	unit time cost to the cloud provider to offer a VM

Table 4.2: Cloud provider model parameters.

as the load λ increases, the region where serverless is cheaper decreases, since a smaller time unit cost is required for serverless to be more cost effective.

4.3 Cloud Services Model: Provider perspective

The previous section analyzed how a user can minimize its cost by leveraging different cloud services to process its load. The reduction in cost attained by the user translates directly to a reduced revenue (and thus, profit) to the cloud provider. Therefore, it becomes essential for the cloud provider to optimally set prices for cloud services to ensure maximum profits.

The cloud provider can clearly influence how a rational client leverages different clouds services, as the provider sets the prices. For instance, if the price for SC is comparable to VMs, a rational client will divert all its load to SC, and never rent a VM (as shown in the previous section). On the other hand, if SC price is much larger than VM price, for almost all loads the client will prefer to use only VMs. Last, there is price range for SC and VM that the client will prefer to leverage both services. Hence, the relative pricing between SC and VM determines the services purchased by a rational client, and consequently the revenue of the provider.

Clearly, a provider incurs costs when offering cloud services, which can depend on the kind of service. These differences in costs are due to differences in hardware and software, management practices, and necessary performance. Thus, the cost of offering a VM to client is different from the cost of offering SC. In particular, let β_v and β_s be the unit time cost for the cloud provider to offer a single VM and a single container for SC, respectively. Table 4.2 summarizes the cloud provider model parameters.

Note that a VM demands more resources from a physical machine than a container used in SC,

which are more lightweight, as shown in the literature [74]. Therefore, a physical server can offer more CPU cycles devoted to the user jobs when leveraging containers. Thus, a provider can pack in more containers on a server as compared to VMs, and this reduces the operating costs for the provider on a per unit basis (hardware costs, power usage etc.). Consequently, the cost of offering a single VM is larger than that of offering a single container for SC, $\beta_s < \beta_v$. Moreover we assume the cost ratio to be inversely proportional to performance ratio, such that $\beta_s/\beta_v = \mu_v/\mu_s$. This is quite useful because it relates the service capacity of SC with its cost in comparison to VM.

Last, since containers can be implemented more efficiently in a given hardware, we assume that $\mu_s > \mu_v$. Thus, the cost to the provider as a function of λ is given by:

$$c_p(\lambda) = \beta_s s(\lambda)/\mu_s + \beta_v v(\lambda) \tag{4.4}$$

where $v(\lambda)$ is the number of VMs rented by the client and $s(\lambda)$ is the client load sent to SC. As SC service is modeled by a M/G/∞ queue we use its expectation to denote the number of containers needed in SC. Thus, $s(\lambda)/\mu_s$ is the expected number of containers required to offer SC for a load of $s(\lambda)$.

The provider profit is simply the difference between its costs and revenue for providing and selling a given service. Note that this revenue is exactly the client cost for using the cloud services, and is given by:

$$p(\lambda) = c(\lambda) - c_p(\lambda) = v(\lambda)(\alpha_v - \beta_v) + s(\lambda)/\mu_s(\alpha_s - \beta_s) \tag{4.5}$$

where $c(\lambda)$ is the cost for the client which is identical to the revenue of the provider, as given by equation 4.1.

Given this formulation, how should a provider set its prices to maximize profits under a rational client that minimizes its costs? In what follows we tackle this problem.

4.3.1 The Provider-Client Game

Consider the problem of a provider setting prices to maximize profits under a client that leverages cloud services to minimize its cost. We model this problem as a two-player game, using a game theoretic formulation [76]. In particular, we consider a sequential game with perfect information where the provider moves first and determines prices for its cloud services. The client moves second and determines the services that will be purchased. In our formulation, the provider must set α_s and α_v (prices) and the client must decide on $v(\lambda)$ and $s(\lambda)$ (number of VMs and load to SC). Note that being a game of perfect information, both players have access to all model parameters and equations, including λ.

We assume the client will purchase cloud services to process its load and meet its performance constraint. Under such consideration, a provider could simply set prices arbitrarily high to increase its profits. However, given a scenario with multiple cloud providers (as is the case in the current market), the client could simply purchase services from another cloud provider, in order to minimize its cost. In order to capture this phenomenon, we assume that the provider sets a fixed price for VM rental, due to strong market competition. In particular, we let $\alpha_v = 1$. Thus, our provider is left with choosing the price α_s for SC.

The solution to this game is a strategy for the provider and a strategy for the client. A Nash equilibrium in this game is a pair of strategies (one for the provider, another for the client) such that neither player can benefit from unilaterally changing strategies. A Nash equilibrium can be found using backward induction in the game tree [76], as follows. For every possible price set by the provider, the client minimizes its cost by allocating its load to VMs and SC. Among all this scenarios, the provider chooses the price that maximizes its profit. In the analysis that follows, we determine this Nash equilibrium.

4.3.2 Analysis

Recall that in §4.2 we determined the minimum cost for the client under a given price. However, recall that the optimal allocation strongly depends on model parameters and has three different

regimes: SC only, SC+VM, and VM only. In what follows, we consider each of these regimes.

Case 1 (SC only): $\rho_t \mu_v \leq \alpha_v \mu_s / \alpha_s$. In this regime, we have that $p(\lambda) = (\lambda/\mu_s)(\alpha_s - \beta_v/\mu_s)$. This profit is maximized by setting α_s as large as possible within the constraint, since the profit grows linearly with α_s. Thus, the maximum value for α_s that satisfy the constraint, denoted by $\alpha_s^{(1)}$, is given by:

$$\alpha_s^{(1)} = \mu_s / \rho_t \tag{4.6}$$

Therefore, $\alpha_s^{(1)} = \mu_s/\rho_t$ is the price that maximizes the profit in this regime. Note that the price is proportional to the capacity of the SC container and inversely proportional to the target utilization set by the user, ρ_t. In particular, a user that has a more conservative performance constraint (e.g., lower target utilization) will observe a higher price, since resources cannot be used as efficiently. The maximum profit in this regime is given by

$$p^{(1)}(\lambda) = \lambda \left(\frac{1}{\rho_t} - \frac{\beta_v}{\mu_s^2} \right) \tag{4.7}$$

Case 2 (SC+VM): $\lambda - \lfloor \lambda / \rho_t \mu_v \rfloor \rho_t \mu_v \leq \alpha_v \mu_s / \alpha_s$. In this regime we have that

$$p(\lambda) = \left\lfloor \frac{\lambda}{\rho_t \mu_v} \right\rfloor (1 - \beta_v) + \left(\frac{\lambda}{\mu_s} - \left\lfloor \frac{\lambda}{\rho_t \mu_v} \right\rfloor \frac{\rho_t \mu_v}{\mu_s} \right) \left(\alpha_s - \frac{\beta_v}{\mu_s} \right) \tag{4.8}$$

Again, this profit increases linearly with α_s and thus is maximized by setting α_s as large as possible within the constraint that determines this case. Thus, the largest possible value for α_s in this case, denoted by $\alpha_s^{(2)}$, is given by:

$$\alpha_s^{(2)} = \frac{\mu_s}{\lambda - \lfloor \lambda / \rho_t \rfloor \rho_t} \tag{4.9}$$

which maximizes the profit in this case. Note that the price is proportional to the capacity of the SC container and inversely proportional to load offered to SC. In particular, a user that sends more load to SC will observe a lower price, in order for the provider to maintain its revenue. The maximum profit in this regime is then given by

$$p^{(2)}(\lambda) = \left\lfloor \frac{\lambda}{\rho_t \mu_v} \right\rfloor (1 - \beta_v) + \left(\lambda - \left\lfloor \frac{\lambda}{\rho_t \mu_v} \right\rfloor \rho_t \mu_v \right) \left(\frac{1}{\lambda - \lfloor \lambda / \rho_t \rfloor \rho_t} - \frac{\beta_v}{\mu_s^2} \right) \tag{4.10}$$

Case 3 (VM only): otherwise. In this case the profit is independent of α_s as only VMs are being used. In particular, if $\alpha_s \geq \alpha_s^{(2)}$ then only VMs will be used by the client. The profit is this regime is given by

$$p^{(3)}(\lambda) = \left(1 + \left\lfloor \frac{\lambda}{\rho_t} \right\rfloor\right)(1 - \beta_v) \tag{4.11}$$

The profit in this region is independent of α_s due to no load being sent to SC. Therefore, the value of α_s can be set arbitrarily large subject to the constraint $\alpha_s \geq \alpha_s^{(2)}$. While the optimal value of α_s is not relevant in this case, it is important when we consider distribution of workloads. Hence, in this region, we assume the optimal value of α_s as follows:

$$\alpha_s^{(3)} = \alpha_s^{(2)} \tag{4.12}$$

Since the provider wants the maximum profit, it can consider the maximum over the three possible regimes, which is given by:

$$p^*(\lambda) = \max\{p^{(1)}(\lambda), p^{(2)}(\lambda), p^{(3)}(\lambda)\} \tag{4.13}$$

Moreover, the price that achieves this optimum profit is given by:

$$\alpha_s^* = \alpha_s^{(o)}, \text{ where } o = \arg\max_{i=1,2,3}\{p^{(i)}(\lambda)\} \tag{4.14}$$

Theorem 1. *Consider the Provider-Client game where the strategy for the provider is a choice for α_s and its payoff given by eq. 4.5 and the strategy for the client is the choice of $v(\lambda)$ (number of rented VMs) and $s(\lambda)$ (load to SC) and payoff given by the negation of eq. 4.1. The choices given by eq. 4.14 (for the provider) and eqs. 4.2 and 4.3 (for the client) is a Nash equilibrium of this game.*

Proof. The proof is by construction of the equations that determine the Nash equilibrium via backward induction on the game tree. Note that for a given price α_s chosen by the provider, the strategy that minimizes the client cost (maximizes its payoff) is given by eqs. 4.2 and 4.3. Thus, the client

cannot reduce its cost (improve its payoff) by deviating from this strategy. Similarly, given the client strategy of minimizing its cost as determined by eqs. 4.2 and 4.3 and the strategy of the provider of determining the price to maximize its profit as in eq. 4.13, the provider cannot improve its payoff by deviating from the price given in eq. 4.14. □

Numerical evaluations

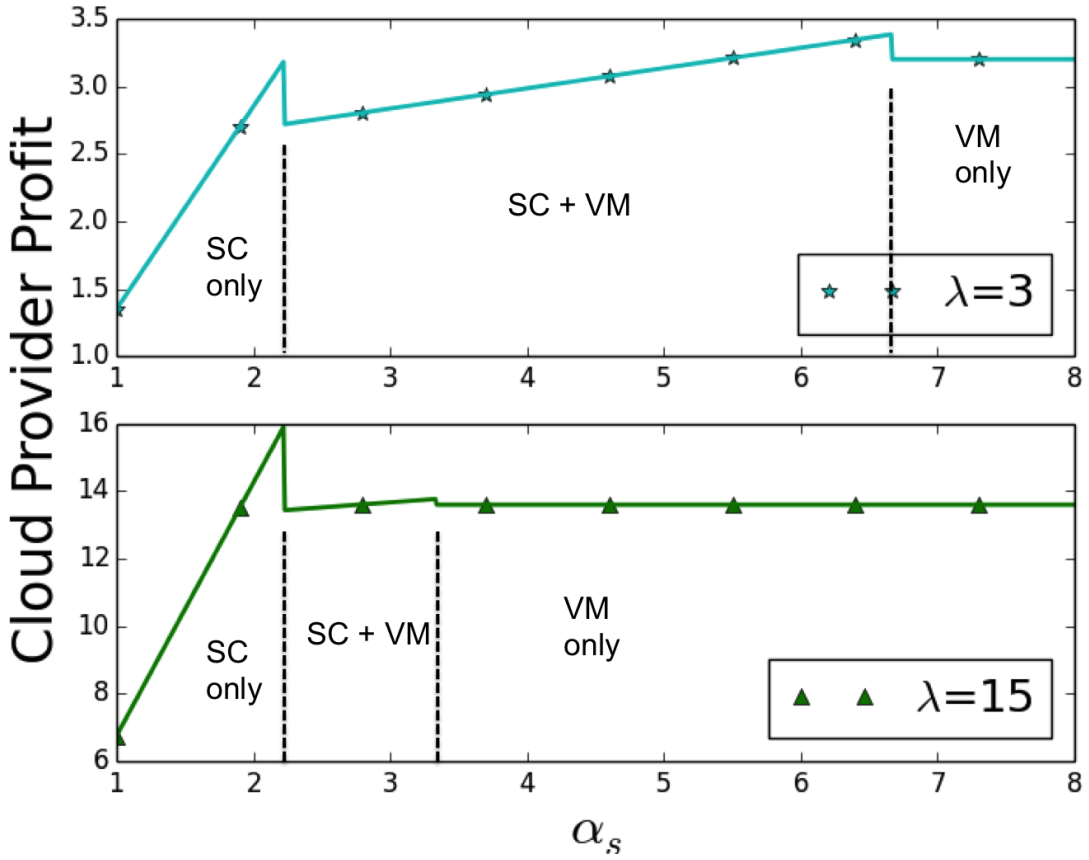


Figure 4.5: Provider profit as a function of SC price, α_s.

Figure 4.5 shows the provider's profit as a function of the SC price, α_s, for two values for load, corresponding to each plot. For both cases, as the price increases we observe three distinct regions, which correspond to the user configuration: SC only, SC+VM, VM only. The first region where $\alpha_s < 2.22$ shows a fast linear increase in the provider's profit since all the workload is served by SC. In this region, the profit rises simply because the user is paying a higher price to use SC for

the same workload. At $\alpha_s = 2.22$, there is a sudden drop in the profit as the user reduces its cost by utilizing the hybrid model SC+VM, with the majority served on the VM and the remaining load on SC. The profit again starts to increase as the price increases because the same load served in SC is now being charged a higher price. For $\lambda = 3$, the profit increases linearly for $\alpha_s \in [2.22, 6.6]$, while for $\lambda = 15$ it increases in the range $\alpha_s \in [2.22, 3.4]$. Note that the profit slope in this range is not as steep as in the SC only regime since the workload on SC is much smaller (only the excess load). As the price continues to increase we observe a second sudden drop in the profit, at $\alpha_s = 6.6$ and $\alpha_s = 3.4$, respectively. This drop occurs because the user can reduce its cost by using a second VM to serve the excess load, not sending any more load to SC. After this point, increasing the price α_s has no bearing on the profit as the user has adopted a VM only configuration. Thus, note that as α_s increases from zero onward, the user reacts accordingly to minimize its cost and this has a direct influence on the provider's profit.

The shape of these two curves is not particular of this parameter configuration. Given our model formulation and the optimal prices and profit established in equations 4.14 and 4.13, there will always be two peaks corresponding to the transition from all SC to a hybrid SC+VM configuration and from the hybrid SC+VM to an all VM configuration. Moreover, since these are peaks, the optimal profit is bound to occur at one of these two peaks. Interestingly, the transition point that provides a higher profit does depend on the parameter configuration, as illustrated in Figure 4.5. However, as λ increases the optimal profit will be attained at the first peak, with the highest possible price within the all SC regime. As λ decreases, the optimal price is attained in the second peak, with the highest possible price within the SC+VM regime. Again, this is illustrated in Figure 4.5.

In this section, we discuss and propose future work in D-TAIL that will be promising and further improve the datacenter network performance.

Partial availability of flow size information. D-TAIL works in either the size-blind mode (flow-size information is not available) or the size-aware mode (flow-size information is available). Given the two extremes in terms of flow-size information available, naturally a question arises whether D-TAIL can have a mode relying on partial availability of flow-size information. The

partial availability of flow-size information can be obtained in two ways. Firstly, as shown in [15], flow-size information can be obtained for certain workloads with a accuracy greater than 80%. In this case, D-TAIL's priority computation can utilize the predicted flow-size information to compute the priority class for the flow. In future work, it will be interesting to explore if the partial information can reduce the tail latency of the transport protocols. Also, identifying the percentage of prediction accuracy required would be beneficial to classify the mode of usage in D-TAIL for a workload. Secondly, a datacenter can have a diverse mix of applications and the application developers for a subset of applications can provide the flow-size information to the transport protocol. It is worth investigating in the future if using a hybrid of size-blind mode and size-aware mode, i.e. using size-blind for flows without flow-size information and using size-aware for flows with flow-size information, can improve the overall performance of the datacenter networks.

Tail-Latency as a function of α. In D-TAIL, the tunable parameter α is powerful because it can be fine-tuned to achieve better performance for a variety of workloads. However, finding the value of α requires the use of Bayesian optimization. In the optimization, multiple iterations of the experiments are required to identify an α that improves performance. While all of our experiments have shown that 25 iterations are sufficient for the analyzed workloads, it is entirely plausible that significantly more iterations can be required for other workloads. We believe that understanding the impact of α on tail-latency would be beneficial. One opportunity to explore is finding tail-latency as a function of α dependent on factors including workload characteristics such as flow-size distribution and flow arrival rate, and network characteristics such as topology, link-speeds, and number of priority classes in switches. Having this function can further enable datacenter operators to easily figure out the value of α for their workloads.

Optimally selecting the bin boundaries in Priority Quantization Priority Quantization is dependent on the number of priority classes supported by the switches. Typically, commodity switches support 4 or 8 priority classes. In priority quantization, we aimed to distribute equal number of bytes across the available priority classes. This distribution required splitting the infinite

range of priority values into 4 to 8 bins depending on the switches. In our simulations assuming 8 bins, we had to identify 7 priority values that would serve as the bin boundaries. We observed that the tail latency for a workload was extremely sensitive to the selection of bin boundaries. Obtaining the bin boundaries that reduced tail-latencies was a challenging task due to the time factor of age of flow in the priority calculation. The age of flow makes it difficult to get an even distribution of bytes across the priority classes. One possible strategy worth exploring in the future work is using Bayesian Optimization for selecting the bin boundaries. This approach adds complexity due to the need for fine-tuning multiple parameters. However, it would immensely simplify the protocol deployment for the datacenter operators.

4.4 Related work

A large body of work has explored different pricing schemes for different cloud services paradigms, considering both the perspective of the customer (how to minimize cost) and the cloud provider (how to maximize revenue or profit) [77, 78, 79, 80]. Cloud providers also offer services under a dynamic pricing scheme (i.e., spot instances), and some recent works have focused on price bidding and price prediction and dynamic VM allocation to reduce costs [81, 82, 83]. A common line of work explores the combination of multiple cloud service paradigms, such as dynamic pricing, fixed pricing, and VMs that can be switched on and off, under different workloads, such as video streaming, and scientific computation, in order to reduce customer costs [84, 80, 85, 86]. Another area of research has focused on minimizing customer costs in hybrid (private/public) cloud providers, especially where they explore provisioning of private clouds and the offloading of excess computations to a public cloud [87, 88]. Although these prior works are similar in scope to the current paper, to the best of our knowledge, none of them have explored a hybrid approach that combines VM rentals and serverless computing (SC) to reduce customer cost and maximize provider profit.

Conclusion

The thesis has tackled a twofold problem of achieving high performance and identifying a pricing strategy for the cloud provider. Specifically, increase in performance is achieved by improving transport protocols in datacenter networks and proposing a container deployment system.

In datacenter networks, the thesis has presented D-TAIL, an easy to implement datacenter transport adaptive mechanism requiring only a minor software change in prioritization-based scheduling in datacenter transport. D-TAIL can be fine-tuned for each workload using a single tunable parameter α to implement a range of scheduling policies like FCFS, enhanced SRPT, and LAS, thereby enabling the datacenter operator to meet a variety of performance objectives as desired. D-TAIL is flexible and can operate in two modes: size-blind when flow size is not available and size-aware for even better performance when flow size information is available. The evaluations show that D-TAIL is robust, can utilize and be compatible with existing transport protocols like DCTCP, pFabric, and Homa. Finally, a key contribution is the use of Bayesian optimization to identify the value of α that maximizes improvement for any workload. In addition, the thesis also presents a container deployment system consisting of peer-to-peer network and virtual file system with content-addressable storage. The proposed deployment design addresses the problem of cold start along with increasing compute availability, and preventing network bottlenecks.

For pricing, the thesis has addressed a hybrid system where a customer can split its workload between VM rental and SC in order to minimize costs while satisfying a given performance constraint. A key finding is the existence of three optimal operational regimes, where depending on system parameters, the Nash equilibrium of the game (i.e., the optimal strategy for the customer and provider) is for the customer to use only SC or only VM or a combination of both. Moreover, we analytically characterize these three regimes as a function of system parameters, such as determining the optimal price for SC. Finally, we believe the proposed framework to study SC pricing

and the various insights provided in this chapter can help both cloud providers and customers better understand the tradeoffs and implications of a hybrid system that combines SC and VM rental with their corresponding pricing structure.

References

[1] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *ACM SIGCOMM*, 2018.

[2] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "Pfabric: Minimal near-optimal datacenter transport," in *ACM SIGCOMM*, 2013.

[3] *Amazon earnings report*, `https://www.marketwatch.com/story/amazon-is-worth-so-much-because-aws-is-techs-true-unicorn-2017-04-27`, Accessed: 2018-04-24, 2017.

[4] E. Protalinski, *Microsoft earnings*, `https://venturebeat.com/2020/04/29/microsoft-earnings-q3-2020/`, 2020.

[5] K. Mahajan, S. Mahajan, V. Misra, and D. Rubenstein, "Exploiting content similarity to address cold start in container deployments," in *ACM CoNEXT*, 2019.

[6] K. Mahajan, D. R. Figueiredo, V. Misra, and D. Rubenstein, "Optimal pricing for serverless computing," in *IEEE Global Communications Conference*, 2019.

[7] J. Maida, *Technavio global data center report*, `https://www.businesswire.com/news/home/20180926005961/en/Global-Data-Center-Market-2018-2022-Increasing-Interest`, 2018.

[8] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik, "Re-architecting datacenter networks and stacks for low latency and high performance," in *ACM Special Interest Group on Data Communication*, 2017.

[9] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," *ACM SIGCOMM CCR*, 2011.

[10] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, "Presto: Edge-based load balancing for fast datacenter networks," in *ACM SIGCOMM*, 2015.

[11] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "Conga: Distributed congestion-aware load balancing for datacenters," in *ACM SIGCOMM*, 2014.

[12] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, 2013.

[13] P. A. Misra, M. F. Borge, I. n. Goiri, A. R. Lebeck, W. Zwaenepoel, and R. Bianchini, "Managing tail latency in datacenter-scale file systems under production constraints," in *Proceedings of the Fourteenth EuroSys Conference*, 2019.

[14] N. Bansal and M. Harchol-Balter, "Analysis of srpt scheduling: Investigating unfairness," in *ACM SIGMETRICS*, 2001.

[15] V. Đukić, S. A. Jyothi, B. Karlas, M. Owaida, C. Zhang, and A. Singla, "Is advance knowledge of flow sizes a plausible assumption?" In *USENIX NSDI*, 2019.

[16] L. Bottou, "Stochastic gradient descent tricks," in *Neural Networks: Tricks of the Trade: Second Edition*, G. Montavon, G. B. Orr, and K.-R. Müller, Eds. 2012.

[17] H. Feng, V. Misra, and D. Rubenstein, "PBS: A unified priority-based scheduler," in *ACM SIGMETRICS Performance Evaluation Review*, 2007.

[18] R. Ricci, E. Eide, and the Cloudlab Team, *Cloudlab*, `https://cloudlab.us/`, 2014.

[19] *Cloud drives changes in network chip architectures*, `https://semiengineering.com/cloud-forcing-big-changes-in-networking-chips/`, 2018.

[20] W. Chen, P. Cheng, F. Ren, R. Shu, and C. Lin, "Ease the queue oscillation: Analysis and enhancement of dctcp," in *IEEE ICDCS*, 2013.

[21] M. Alizadeh, A. Javanmard, and B. Prabhakar, "Analysis of dctcp: Stability, convergence, and fairness," in *ACM SIGMETRICS*, 2011.

[22] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and W. Sun, "Pias: Practical information-agnostic flow scheduling for data center networks," in *ACM HotNets*, 2014.

[23] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker, "Phost: Distributed near-optimal datacenter transport over commodity network fabric," in *ACM CoNEXT*, 2015.

[24] J. Močkus, "On bayesian methods for seeking the extremum," in *Optimization Techniques IFIP Technical Conference Novosibirsk*, 1974.

[25] *Pfabric ns2 network simulator*, `https://github.com/camsas/qjump-ns2`, 2015.

[26] *Homa omnet++ network simulator*, `https://github.com/PlatformLab/HomaSimulation`, 2019.

[27] *Ns3 network simulator*, `https://www.nsnam.org/`, 2019.

[28] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Performance Evaluation Review*, 2012.

[29] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *ACM SIGCOMM CCR*, 2015.

[30] S. Bakshi, *Ns3 dctcp implementation*, `https://github.com/ShikhaBakshi/ns-3-dev-git/commits/dctcp`, 2018.

[31] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *ACM Special Interest Group on Data Communication*, 2015.

[32] L. Torvalds, *Kernel ebpf repositories*, `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/uapi/linux/bpf.h`, 2019.

[33] *Bpf classifier and actions in tc*, `http://man7.org/linux/man-pages/man8/tc-bpf.8.html`, 2019.

[34] D. EMC, *Dell system s4048-on switch*, `https://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/Dell-EMC-Networking-S4048-ON-Spec-Sheet.pdf`, 2019.

[35] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks.," in *USENIX NSDI*, 2010.

[36] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized zero-queue datacenter network," *ACM SIGCOMM CCR*, 2015.

[37] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic flow scheduling for commodity data centers," in *USENIX NSDI*, 2015.

[38] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues don't matter when you can jump them!" In *USENIX NSDI*, 2015.

[39] 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 *USENIX NSDI*, 2012.

[40] C.-Y. Hong, M. Caesar, and P Godfrey, "Finishing flows quickly with preemptive scheduling," in *ACM SIGCOMM*, 2012.

[41] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid task provisioning with serverless-optimized containers," in *USENIX Annual Technical Conference*, 2018.

[42] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*, S. Chaudhary, G. Somani, and R. Buyya, Eds. 2017.

[43] *Cold starts in aws lambda*, https://mikhail.io/serverless/coldstarts/aws/, 2019.

[44] *Cisco*, https://www.cisco.com/, 2019.

[45] Cisco, *Cisco white paper*, https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html, 2018.

[46] *Apache openwhisk: Https://github.com/apache/incubator-openwhisk*, 2017.

[47] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *USENIX HotCloud*, 2016.

[48] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *IEEE (ICDCSW)*, 2017.

[49] E. Jonas, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *ACM Symposium on Cloud Computing (SoCC)*, 2017.

[50] M. Yan, P. Castro, P. Cheng, and V. Ishakian, "Building a chatbot with serverless computing," in *ACM International Workshop on Mashups of Things and APIs (MOTA)*, 2016.

[51] M. Crane and J. Lin, "An exploration of serverless architectures for information retrieval," in *ACM SIGIR International Conference on Theory of Information Retrieval (ICTIR)*, 2017.

[52] Q. Jiang, Y. C. Lee, and A. Y. Zomaya, "Serverless execution of scientific workflows," in *Service-Oriented Computing*, 2017.

[53] C. Zhang, M. Yu, W. Wang, and F. Yan, "Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving," in *USENIX Annual Technical Conference*, 2019.

[54] *Aws lambda*, https://aws.amazon.com/lambda/, 2019.

[55] *Google cloud functions*, https://cloud.google.com/functions/, 2019.

[56] *Microsoft azure functions*, https://azure.microsoft.com/en-us/services/functions/, 2019.

[57] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *USENIX NSDI*, 2019.

[58] *Amazon s3*, https://aws.amazon.com/s3/, 2019.

[59] R. Merkle, "A digital signature based on a conventional encryption function," 1987.

[60] J. Reich, O. Laadan, E. Brosh, A. Sherman, V. Misra, J. Nieh, and D. Rubenstein, "Vmtorrent: Scalable p2p virtual machine streaming," in *ACM CoNEXT*, 2012.

[61] *Criu live migration*, https://criu.org/Live_migration, 2019.

[62] T. K. Authors, *Kubernetes*, https://kubernetes.io/, 2019.

[63] *Kubernetes: Scaling an application*, https://cloud.google.com/kubernetes-engine/docs/how-to/scaling-apps, 2019.

[64] *Aws lambda function versioning and aliases*, https://docs.aws.amazon.com/lambda/latest/dg/versioning-aliases.html, 2019.

[65] *Prestashop*, https://github.com/PrestaShop/docker, 2019.

[66] *Python package index*, https://pypi.org/, 2019.

[67] *Pandas: Python data analysis library*, https://pandas.pydata.org/, 2019.

[68] J. Benet, "IPFS - content addressed, versioned, P2P file system," *CoRR*, 2014.

[69] *T2 micro instance*, https://aws.amazon.com/ec2/instance-types/t2/, 2019.

[70] *Aws*, https://aws.amazon.com, 2019.

[71] *Apache spark*, https://spark.apache.org/, 2019.

[72] *Rsync*, https://linux.die.net/man/1/rsync, 2019.

[73] R. Byrro, *Can we solve serverless cold starts?* https://dashbird.io/blog/can-we-solve-serverless-cold-starts/, 2019.

[74] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and virtual machines at scale: A comparative study," in *ACM International Middleware Conference*, 2016.

[75] L. Kleinrock, *Theory, Volume 1, Queueing Systems*. 1975.

[76] M. J. Osborne and A. Rubinstein, *A course in game theory*. 1994.

[77] D. Kumar, G. Baranwal, Z. Raza, and D. P. Vidyarthi, "A survey on spot pricing in cloud computing," *Journal of Network and Systems Management*, 2017.

[78] B. Jennings and R. Stadler, "Resource management in clouds: Survey and research challenges," *Journal of Network and Systems Management*, 2015.

[79] N. C. Luong, P. Wang, D. Niyato, Y. Wen, and Z. Han, "Resource management in cloud networking using economic analysis and pricing models: A survey," *IEEE Communications Surveys & Tutorials*, 2017.

[80] I. Menache, O. Shamir, and N. Jain, "On-demand, spot, or both: Dynamic resource allocation for executing batch jobs in the cloud," in *USENIX ICAC*, 2014.

[81] P. Sharma, D. Irwin, and P. Shenoy, "How not to bid the cloud," in *USENIX HotCloud*, 2016.

[82] S. Subramanya, T. Guo, P. Sharma, D. Irwin, and P. Shenoy, "Spoton: A batch computing service for the spot market," in *ACM Symposium on Cloud Computing*, 2015.

[83] H. Xu and B. Li, "Dynamic cloud pricing for revenue maximization," *IEEE TCC*, 2013.

[84] D. J. Dubois and G. Casale, "Optispot: Minimizing application deployment cost using spot cloud resources," *Cluster Computing*, 2016.

[85] C. Wang, B. Urgaonkar, A. Gupta, G. Kesidis, and Q. Liang, "Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud," in *EuroSys*, 2017.

[86] Z. Xu, C. Stewart, N. Deng, and X. Wang, "Blending on-demand and spot instances to lower costs for in-memory storage," in *IEEE INFOCOM*, 2016.

[87] R. Buyya, S. Pandey, and C. Vecchiola, "Cloudbus toolkit for market-oriented cloud computing," in *IEEE CLOUD*, 2009.

[88] M. Malawski, K. Figiela, and J. Nabrzyski, "Cost minimization for computational applications on hybrid cloud infrastructures," *Future Generation Computer Systems*, 2013.