Ether: Providing both Interactive Service and Fairness in Multi-Tenant Datacenters

Mojtaba Malekpourshahraki
University of Illinois at Chicago
mmalek3@uic.edu

Brent Stephens
University of Illinois at Chicago
brents@uic.edu

Balajee Vamanan
University of Illinois at Chicago
bvamanan@uic.edu

ABSTRACT

Multi-tenant datacenters and cloud networks must provide both isolation and interactive service to tenant applications, many of which are sensitive to tail flow completion times. Network operators must also ensure high utilization of network capacity to reduce cost. Existing approaches that *statically* partition network capacity, in either time or space, provide good isolation but suffer from under-utilization. Existing schemes that dynamically allocate capacity to tenants incur either decreased fairness or high tail flow completion times. To overcome these limitations, we propose *Ether*. *Ether* is able to overcome these limitations because it can prioritize bursty flows during short congestion episodes while still ensuring fairness at long timescales. In this paper, we present a preliminary design of *Ether* and discuss its feasibility in today's programmable switches. Our evaluations show that, at high loads, *Ether* achieves 23% improvement in tail flow completion times (FCT) when compared with idealized fair queueing (FQ) while still providing similar fairness as FQ. In contrast, pFabric, which optimizes FCT, worsens fairness by a factor of 1.8 when compared with *Ether*.

CCS CONCEPTS

• **Networks** → **Packet scheduling**; **Data center networks**; *Packet classification.*

KEYWORDS

Datacenter, Fairness, Optimal FCT, Multi-tenant datacenter

APNet '19, August 17–18, 2019, Beijing, China
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7635-8/19/08…$15.00
https://doi.org/10.1145/3343180.3343187

ACM Reference Format:
Mojtaba Malekpourshahraki, Brent Stephens, and Balajee Vamanan. 2019. *Ether*: Providing both Interactive Service and Fairness in Multi-Tenant Datacenters. In *3rd Asia-Pacific Workshop on Networking 2019 (APNet '19), August 17–18, 2019, Beijing, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3343180.3343187

1 INTRODUCTION

Modern cloud datacenters host several tenants with diverse application characteristics [16, 17]. To meet the stringent Service Level Agreements (SLAs), network operators must isolate tenant applications and provide *fairness* among tenants within the network. Further, the SLAs for some applications require tight bounds on *network delay* (e.g., 99th percentile flow completion times must be less than 2–30 ms [25]). Finally, it is important for operators to ensure that the overall network *utilization* is high [18].

Unfortunately, today's networks struggle to simultaneously provide multi-tenant isolation, minimize Flow Completion Times (FCT), and drive high utilization [10]. Many cloud datacenters isolate tenants by statically partitioning the network resources [1, 20]. However, such approaches are not work conserving and reduce resource utilization. Although a few recent proposals provide good fairness and high resource utilization, they do not minimize tail FCTs [9, 17, 21, 24]. Similarly, while there has been a significant focus on reducing tail FCTs in datacenter networks to meet the requirements of Online Data Intensive (OLDI) applications (e.g., Web Search), most existing systems for doing so do not provide multi-tenant isolation [6, 8, 19, 23, 27]. One way of providing interactive performance guarantees to tenants is to limit the burst size of applications, in addition to rate (bandwidth) limits [16]. However, many OLDI applications cause synchronized bursts, known as *incasts* and limiting the burst size of those applications would adversely affect the overall response time [25].

Our goal in this paper is to improve the tail (e.g., 99th) Flow Completion Time (FCT) of tenant applications while providing isolation (fairness) between tenants, without imposing limits on traffic bursts or limiting network throughput. We achieve this goal by relying on the following *key insights*: (1) Fairness is a relatively long-term concern, and it is often sufficient to enforce fairness at a coarse time granularity; (2) Incast and minimizing flow completion times are problems that occur at very small timescales. For example, a recent study found that most congestion events are shorter than a single RTT [28].

We present the design of *Ether*, a system that leverages the two preceding insights. Crucially, *Ether* includes two building blocks: a *fairness optimizer*, which ensures that tenants share the bottleneck capacity in a fair manner, and a *tail optimizer*, which implements Least Slack Time First (LSTF) scheduling among flows within short intervals. Our key contributions include the following:

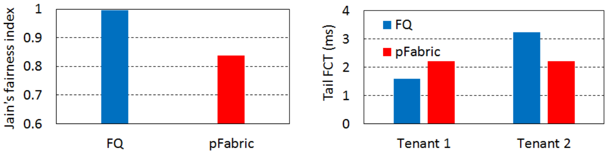
- We present the design of *Ether*, which prioritizes critical flows (e.g., short flows or flows with least slack) during short congestion episodes such as incasts while ensuring fairness over longer timescales.
- We discuss the feasibility of *Ether* in the context of multiple of today's programmable switches and other network devices and argue that the design of *Ether* is implementation friendly.
- Using ns-3 simulations [2], we show that *Ether* achieves 23% improvement in tail FCT while providing similar fairness as fair queuing (ideal), whereas pFabric, which optimizes FCT, worsens fairness by a factor of 1.8.

The paper is organized as follows. We start with an opportunity study and motivate our work in Section 2. We analyze our design, provide guidance for setting the design parameters based on broad workload characteristics, and we provide a brief sketch of an implementation using programmable switches in Section 3. We present detailed evaluations that show bottomline performance, isolation of our techniques, and parameter sensitivity in Section 4. Finally, we conclude with open questions and future work in Section 6.

2 MOTIVATION

Table 1: A summary of existing approaches to network isolation or reducing tail flow completion times (Tail FCT). A * implies that there are limitations to the design that are discussed in Section 5

		Network Isolation	Tail FCT Reduction
pFabric	[6]		✓
PIAS	[8]		✓
UPS	[19]		✓
EyeQ	[17]	✓	
AFQ	[24]	✓	
Silo	[16]	✓	
pHost *	[13]	✓	✓
Trinity *	[14]	✓	✓
Utopia *	[26]	✓	✓
Ether		✓	✓



(a) Jain's fairness index **(b) 99 percentile FCT**

Figure 1: FQ vs. pFabric.

In today's datacenters, it is important to both fairly share the network and minimize tail FCTs. For fairness, recent systems for multi-tenant datacenters either use some form of bandwidth reservations [16, 17] or employ some form of Fair Queuing (FQ) at switches [24] for bandwidth isolation (fairness) among tenants. Unfortunately, neither of these approaches reduce tail FCTs, and reservations can hurt overall utilization. For applications that are sensitive to Flow Completion Times (FCT), recent work has used Shortest Job First (SJF) and Least Slack Time First (LSFT) scheduling because these approaches have been shown to reduce average FCTs [6, 8, 13, 19, 23]. However, these approaches can also be prone to extreme unfairness, especially in situations when competing tenants have different messages sizes or are trying to game the system.

Ideally, it is desirable to both provide isolation among tenants and reduce flow completion times for application flows. Unfortunately, as Table 1 summarizes, no existing projects are able to successfully accomplish both of these goals without significant limitations. When tenants have diverse flow distributions, as is often the case, approaches that optimize FCT favor tenants (applications) with a higher number of short flows and violate fairness. While FQ provides fairness, it hurts FCT because of its inflexibility to prioritize short flows during *short* periods of intermittent congestion (e.g., incasts).

To demonstrate these problems with existing systems, we performed an experiment with ns-3 [2] to compare systems that use fair queuing to isolate tenants with those that use scheduling/prioritization to improve FCT. In this experiment, there are two tenants, and we use idealized FQ to represent systems that provide multi-tenant fairness and pFabric to represent systems that use prioritization to improve FCT. The second tenant generates three times more short flows than the first tenant while both tenants generate the same number of long flows. Figure 1(a) shows the Jain's fairness index [15] among the two tenants, and Figure 1(b) shows the 99^{th} percentile FCT of the two tenant flows. This figure shows that that pFabric's improved FCT comes at the cost of reduced fairness while FQ improves fairness at the cost of FCT.

It is desirable to provide the benefits of both FQ and improved FCT. However, it is non-trivial to combine existing

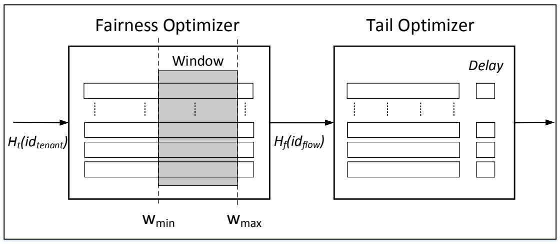


Figure 2: *Ether*'s architecture

approaches that provide either fairness or improved FCT but not both. For example, a straightforward solution to providing both fairness and improved FCT is to employ FQ scheduling among tenants and LSFT or SJF scheduling among flows within each tenant, in a hierarchical fashion. However, this approach would require a prohibitively large number of queues (e.g., With 32 queues at each level, we require 32 $*$ 32 queues). Further, FQ is inflexible and does not allow for prioritizing critical flows during periods of congestion.

To overcome these challenges, we make the *key insight* that fairness is a relatively long-term concern and it is sufficient to enforce fairness at a coarse time granularity. This insight enables us to prioritize critical flows (e.g., short flows or flows with least slack) during short periods of intermittent congestion (e.g., incast). Building on this insight, we propose *Ether* and discuss our design in the following section.

3 DESIGN

3.1 Overview

Instead of hierarchically composing FQ and LSTF schedulers, which would require a large number of queues, *Ether* decouples the enforcement of fairness among tenants and the FCT optimization of tenant flows. Specifically, *Ether* enforces fairness at a coarse granularity of *windows* of tenant packets, whereas packets belonging to different tenants within the same window are scheduled purely based on their critically (e.g., slack).

Figure 2 shows the high-level overview of *Ether*. *Ether* is composed of two stages: *fairness optimizer* and *tail optimizer*. The fairness optimizer fetches the tenant ID and maps the packet to one of its queues; we derive the number of queues required to avoid hash collision in Section 3.2.3. Next, a *window* of packets from each queue is transferred to the tail optimizer queues by hashing on the flow ID (e.g., five tuples). It is important to note that while fairness optimizer hashes using tenant IDs, tail optimizer hashes based on flow IDs. The tail optimizer finally picks the queue to dequeue based on slack. By maintaining the *invariant* that the next window of packets is fetched from the fairness optimizer into the tail optimizer *only* after *all* the packets in the previous window have been dequeued, *Ether* provides fairness among tenants. However, within a window, packets are scheduled purely based on slack, thereby optimizing FCT. In other words, we sacrifice some fairness and allow tenants that have critical flows to steal bandwidth from tenants that do not have critical flows only *within* a window; we enforce fairness among tenants across windows. Because our tail optimizer needs to track distinct flows only within a window, which is much smaller than the total number of flows, we require only a handful number of queues (i.e., not one queue per-flow). In the following sections, we describe our fairness optimizer and tail optimizer in detail.

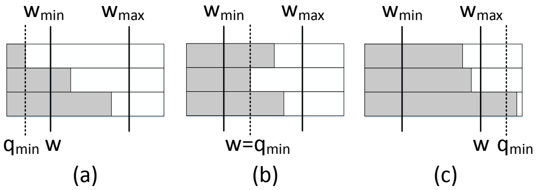


Figure 3: Different conditions for fairness queue set.

3.2 Fairness optimizer

When a packet arrives at the switch, the fairness optimizer fetches the tenant ID and maps the incoming packet to one of its queues by hashing on tenant ID (see Figure 2). The fairness optimizer transfers a *window* of packets to the tail optimizer in rounds. In each round, it dequeues packets from each queue and maps it to one of the tail optimizer queues by hashing on flow ID.

3.2.1 Window size. To provide fairness among tenants, we must dequeue an equal number of packets (i.e., window size of w) from *each* tenant queue at the fairness optimizer. Further, to provide opportunity for the tail optimizer in the next stage, we must choose the largest window size that allows us to dequeue an equal number of packets from every queue. Thus, setting the window size to equal the minimum queue length (q_{min}) allows us to maximize opportunity for the tail optimizer without violating fairness (see Figure 3b). But, there are two other corner cases:

(1) If some, not all, of the tenant queues are (near) empty because they do not have enough data to send, allowing a window size larger than the minimum queue length would provide sufficient opportunity for the tail optimizer without violating fairness (see Figure 3a). To handle this case, we have a configurable lower bound, w_{min}.
(2) If all tenant queues are close to being full, setting the window size to equal the minimum queue length, which is quite large, would overwhelm the tail optimizer with a large number of packets (flows) and may cause rampant hash collisions (see Figure 3c). To handle this case, we have a configurable upper bound, w_{max}.

Putting it all together, equation 1 shows the window size (w) as a function of the minimum queue length (q_{min}), the upper bound of the window size (w_{max}), and the lower bound of the window size (w_{min}).

$$w = \min(w_{max}, \max(w_{min}, q_{min})) \tag{1}$$

Therefore, by having a *dynamic* window size, *Ether* is able to prioritize short flows' by increasing the window size during during short periods of congestion but quickly reduces the window size as congestion subsides to provide fairness.

3.2.2 Window boundaries. Recall from Section 3.2.1 that w_{max} prevents the window from becoming too large and cause hash collisions in the tail optimizer. We now derive the relationship between the window bounds and the number of queues to avoid hash collisions. Assuming that the packet size is s bytes, average flow size is $E[S]$, and number of queues at the fairness optimizer is n_f, then the total number of flows contained in the window w is given by $\frac{n_f \times w \times s}{E[S]}$. To minimize hash collisions at the tail optimizer, the total number of flows contained in the window w should not exceed the number of queues at the tail optimizer (n_t). Thus, we have the following equation:

$$\frac{n_f \times w \times s}{E[S]} \leq n_t \tag{2}$$

$$\implies w \leq \frac{E[S] \times n_t}{s \times n_f} \implies w_{max} = \frac{E[S] \times n_t}{s \times n_f} \tag{3}$$

Equation 3 provides guidance on how to set w_{max}, given the number of queues in fairness optimizer and tail optimizer.

3.2.3 Number of queues. While we require $w_{min} \geq 1$ to guarantee forward progress, in practice, we set w_{min} to be a larger value. Because w_{max} is greater than or equal to w_{min} by definition, we have the following condition for the number of queues:

$$w_{max} \geq w_{min} \implies \frac{E[S] \times n_t}{s \times n_f} \geq w_{min} \tag{4}$$

$$\frac{n_t}{n_f} \geq \frac{s}{E[S]} w_{min} \tag{5}$$

Equation 5 is intuitive and provides guidance on how to pick the number of queues in fairness optimizer and tail optimizer. We require more queues at the tail optimizer if either the average flow size ($E[S]$) is small or if w_{min} is large because we will transfer a large number of flows from fairness optimizer to tail optimizer.

3.3 Tail Optimizer

In each round, *Ether* dequeues all packets within the window from the fairness optimizer to the tail optimizer's queues. Another hash function ($H_t(.)$) is used to map the packets to the tail optimizer's queues. The tail optimizer emulates LSTF scheduling. Instead of slack, each packet contains the queue length, a proxy for delay, that the packet incurred in previous hops. Because slack is defined as the difference between deadline and delay, LSTF scheduling is equivalent to scheduling the packet with the largest delay as flow deadlines are often unknown [11, 19, 23].

Ether updates the slack in packet header during the fairness optimizer's packet enqueue. Our estimation of slack is the total number of packets that gets service before the current packet. Equation 6 calculates an estimation of the total number of packets before enqueuing a packet to the queue i.

$$slack = q_{min} \times (n_f - 1) + S_i + S_t \tag{6}$$

In which S_i is the size of the queue i, and S_t is the total queue size of the tail optimizer.

Ether keeps a delay value for each queue (flow) in the tail optimizer (see delay field in figure 2). During the packet enqueue, *Ether* adds the slack value of the packet to the queue delay. Similarly, during the packet dequeue, the delay of the queue reduces by the slack value of the packet.

Tail optimizer dequeues packets with the maximum delay first. In each round, instead of finding the maximum slack among all queues, which is an operation with $O(log(n))$, we keep track of the maximum slack in a separate temporarily variable and update the changes during enqueue and dequeue.

3.4 Implementation

Ether has the following implementation challenges: (1) tracking the dynamic control parameters (e.g., window boundaries); (2) mapping the incoming packets to the corresponding queues; and (3) two levels of the queue set in the data path. Fortunately, the programmable switches provide features that enable us to address these challenges. Configurable switches provide meta-data to keep the dynamic variables. For the mapping tenant/flow id, p4 provides a wide range of hashing algorithm that could map header fields to the queue-id. PSA does not support two layers of queue sets; however, it is possible to emulate the same behavior by *clone from egress to egress* (CE2E), described in [3]. CE2E provides a line-rate, packet resubmit feature to the egress pipelines. PSA could resubmit packets back to the single queue set, instead of using two levels of queues in the path of packets. We are exploring the implementation of our proposal on programmable switches for our future extended version of *Ether*.

4 EVALUATION

4.1 Methodology

4.1.1 Topology and Workload. We evaluate the *Ether*'s overall performance using ns-3 simulator [2]. We simulate a leaf-spine datacenter topology with 400 servers, 10 spine switches, and 20 leaf switches, connected using 10 Gbps links. The unloaded RTT of the longest path with 4 hops is 80 μs. Our workload, based on existing empirical studies [7, 24], generates a heavy-tailed flow distribution. Specifically, we consider 16 KB *short* flows and 1 MB *long* flows, with long flows, which are much fewer than short flows, contributing to a larger fraction (e.g., 80%) of network load. We also model incast traffic, with the average fan-in degree of 32.

4.1.2 Compared schemes. Because existing schemes optimize either fairness or tail FCT but not both, we compare *Ether* to two strong baselines, which optimize either fairness or tail FCT. We use *Fair Queuing (FQ)* as the baseline for fairness and *pFabric* as the baseline for tail FCT. Switches in our FQ implementation perform per-packet round robin scheduling among tenant flows and the end hosts use DCTCP [5]. Our pFabric implementation implements the shortest job first scheduler at the switches and uses the end host congestion control as presented in their paper [6]. While FQ achieves close-to-ideal ideal fairness, pFabric is the current state-of-the-art for optimizing FCT.

4.1.3 Parameters. *Ether* has four main parameters: (1) n_f, the number of queues in fairness optimizer; (2) n_t, the number of queues in tail optimizer; (3) w_{max}, the maximum possible window; and (4) w_{min}, the minimum possible window size in each round. Because 16–32 queues is typical, we use $n_f = n_t = 16$. We choose $w_{min} = 1$ (limit from Section 3.2.3) and $w_{min} = 570$ (Equation 3).

4.1.4 Comparison metrics. First, we show our bottomline performance by comparing *Ether*'s fairness and 99th percentile FCT with systems that are known to optimize fairness and tail FCT. Then, we perform sensitivity studies on the number of queues — the number of queues in fairness optimizer affects fairness, whereas the number of queues in tail optimizer affects FCT. Finally, we quantify our sensitivity to workload, specifically to the fraction of short flows. We leave real implementation, evaluation using real applications, and exhaustive quantitative evaluations to future work.

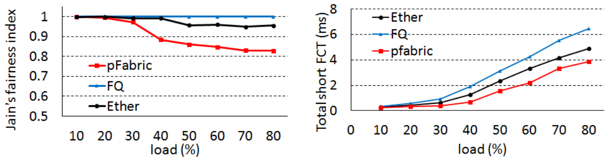


Figure 4: Fairness and tail FCT of *Ether*

4.2 Bottomline performance

Figure 4 compares the bottomline performance of the three schemes (i.e., FQ, pFabric, and *Ether*), in terms of both fairness and tail FCT. In this experiment, we have 10 tenants, each generating a mix of short and long flows as per our workload model (see Section 4.1.1).

Figure 4(a) shows the Jain's Fairness Index (JFI) among tenants (i.e., we aggregate the throughput per tenant and compute JFI among them) for different loads, along X axis. A JFI value of 1 is ideal and fairness decreases as values get smaller than 1. Because our implementation of FQ is ideal, it achieves perfect fairness. Fairness of pFabric suffers as load increases. For loads higher than 60%, pFabric's fairness suffers by about 20%. *Ether* achieves better fairness than pFabric. Specifically, *Ether* achieves similar fairness as ideal (within 5%) and outperforms pFabric by about 18% (JFI of 0.95 for *Ether* vs. 0.8 for pFabric). While our fairness optimizer ensures each tenant gets a fair share of capacity during each window by transferring an equal number of bytes from the tenants' queues, it also provides opportunity to the tail optimizer to improve FCT, which we show next.

Figure 4(b) shows the tail FCT across all tenant flows. While all schemes perform well at low loads, their performance starts to diverge as load increases. At high network load, *Ether* outperforms the FQ and performs closer to pFabric. Because FQ does not consider criticality of flows (e.g., slack), tail packets from short flows incur additional queuing due to long flows. pFabric and *Ether* dequeue packets with the largest slack and improve tail. In summary, unlike pFabric, which improves tail FCT at the cost of worsened fairness, *Ether* improves the tail FCT while achieving *close-to-ideal* to fairness.

4.3 Sensitivity to number of queues

Today's commercial switches support a limited number of queues. We study the effect of the number of queues in fairness optimizer and tail optimizer on the performance of *Ether* using targeted experiments. In each experiment, we fix the number of queues to 16 for one module and vary the number of queues in the other module. In these experiments, we simulate multiple tenants, and each tenant generates a mix of short and long flows. All other parameters are set as per Section 4.1.3.

We study the sensitivity of *Ether*'s fairness to the number of queues in fairness optimizer in Figure 6(a). Because the

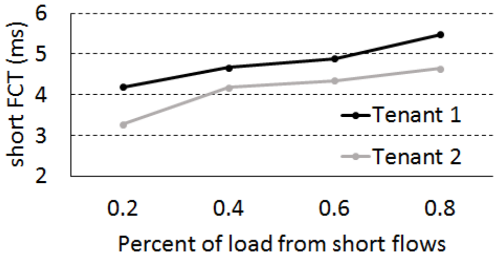


Figure 5: Sensitivity to short flows

queues in fairness optimizer is sensitive to the number of tenants, we simulate a large number of tenants and vary the number of queues in the fairness optimizer. We observe that the performance improves with the number of queues, as expected. While the experiment proves that the number of queues in fairness optimizer is proportional to the number of tenants, we will explore optimizations to reduce the number of queues in our future work.

Figure 6(b) shows the sensitivity of *Ether*'s 99th percentile FCT to the number of queues in tail optimizer. Because the number of queues in the tail optimizer is independent of the number of tenants, we simulate only two tenants. When the number of queues is small, collision among flows increases and the tail optimizer performs sub-optimally. However, as the number of queues increases, the tail optimizer is able to distinguish the flows and schedule them based on least slack. As such, we achieve optimal performance for 16–32 queues.

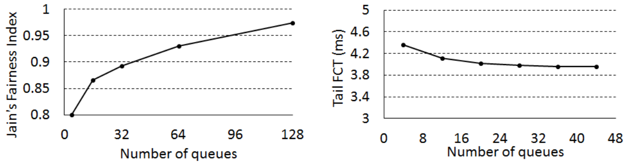
4.4 Sensitivity to short flows

Ether is sensitive to the (average) flow size and flow mix because of our windowing idea. For this study, we simulate two tenants. Figure 5 shows the effect of the traffic pattern (i.e., fraction of the overall load from short flows, shown on X axis) on the 99th percentile FCT of the two tenants. We vary the fraction of short flows but we keep the total network load at 80%. As the fraction of short flows increases, there is increased burstiness (i.e., $q_{min} increases$). Further, the number of distinct flows that fit within a window increases as the fraction of short flows increases, which leads to increased collision at the tail optimizer (see Figure 6b). As a result, the tail FCT increases with the fraction of short flows, for both tenants. Fortunately, the heavy tailed nature of datacenter workloads imply that a large fraction of the overall network load comes from a few long flows (i.e., the typical operating point is towards the left side of X axis), in which *Ether* achieves shorter tail FCT with a handful number of queues.

5 RELATED WORK

Our work is closely related to existing work on improving flow completion times and network isolation. Several existing papers employ Shortest job first (SJF) and least slack-time first (LSTF) scheduling, both at the end hosts and in-network, to optimize flow completion times. pFabric [6] and PIAS [8] schedule packets based on their deadlines or flow sizes to optimize flow completion times; we have extensively discussed and evaluated pFabric [6]. UPS [19] performs LSTF scheduling to reduce the tail flow completion times. DeTail [27] leverages PFC to reduce network queuing delay. While these approaches provide vast improvements in flow completion times, they do not provide good isolation and often favors tenants with a larger fraction of short flows.

Existing work on network isolation provide good fairness but do not explicitly optimize for the tail flow completion times. EyeQ [17] provides bandwidth sharing between tenants by employing a combination of rate limiting and Rate Control Protocol [12]. Silo [16] provides stronger delay guarantees for tenants by limiting the burst size of tenant flows, in addition to rate limiting. AFQ [24] approximates provides per-flow fair queuing but it is not easily extensible to multi-tenant systems. However, unlike *Ether*, EyeQ, Silo, and AFQ do not leverage short periods of intermittent burstiness to optimize tail FCT. Both pHost [13] Utopia [26] optimize flow and coflow completion times, respectively, as well as provide support for multi-tenancy. However, they assume that congestion happens *only* at the network edge, and therefore, require *non-blocking* network, which is prohibitively expensive; existing networks use over-subscription factors of 4–8 [4, 22]. Further, Utopia needs the status of all links. Trinity [14] uses ECN marks to calculate the rate for each VM-2-VM channel to achieve full utilization of bottleneck links, and also provides work conservation and bandwidth guarantees. However, Trinity's reaction to incast is slow as it relies on ECN marks [23]. Other centralized coflow scheduling approaches require real-time link status to schedule coflows, and, therefore do not scale well to large datacenters [14, 26].



(a) Fairness optimizer **(b) Tail optimizer**

Figure 6: Sensitivity to number of queues

6 CONCLUSION

We presented *Ether*, which prioritizes critical flows during short periods of congestion and ensures that such prioritization does not affect fairness. Our design is light-weight and is implementable in today's programmable switches. Our evaluations show that *Ether* achieves both high fairness and low queuing (i.e., low 99^{th} percentile FCT).

REFERENCES

[1] [n.d.]. Amazon Virtual Private Cloud. https://aws.amazon.com/vpc.

[2] [n.d.]. NS-3 network simulator. http://www.nsnam.org/.

[3] [n.d.]. Portable Switch Architecture (PSA). https://p4.org/specs/.

[4] Mohammad Al-Fares et al. 2008. A scalable, commodity data center network architecture. In *SIGCOMM*.

[5] Mohammad Alizadeh et al. 2010. Data Center TCP (DCTCP). In *SIGCOMM*.

[6] Mohammad Alizadeh et al. 2013. pfabric: Minimal near-optimal datacenter transport. In *SIGCOMM*.

[7] Mohammad Alizadeh et al. 2014. CONGA: Distributed congestion-aware load balancing for datacenters. In *SIGCOMM*.

[8] Wei Bai et al. 2015. Information-agnostic Flow Scheduling for Commodity Data Centers. In *NSDI*.

[9] Hitesh Ballani et al. 2011. Towards predictable datacenter networks. In *SIGCOMM*.

[10] Mosharaf Chowdhury et al. 2016. {HUG}: Multi-Resource Fairness for Correlated and Elastic Demands. In *NSDI*.

[11] David D Clark et al. 1992. Supporting real-time applications in an integrated services packet network: Architecture and mechanism. In *SIGCOMM*.

[12] Nandita Dukkipati et al. 2005. Processor Sharing Flows in the Internet. In *IWQoS*.

[13] Peter X Gao et al. 2015. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *CoNEXT*.

[14] Shuihai Hu, Wei Bai, Kai Chen, Chen Tian, Ying Zhang, and Haitao Wu. 2018. Providing bandwidth guarantees, work conservation and low latency simultaneously in the cloud. *IEEE Transactions on Cloud Computing* (2018).

[15] Rajendra K Jain et al. 1984. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation* (1984).

[16] Keon Jang et al. 2015. Silo: Predictable Message Latency in the Cloud. In *SIGCOMM*.

[17] Vimalkumar Jeyakumar et al. 2013. EyeQ: Practical network performance isolation at the edge. In *NSDI*.

[18] David Lo et al. 2015. Heracles: Improving Resource Efficiency at Scale. In *ISCA*.

[19] Radhika Mittal et al. 2016. Universal packet scheduling. In *NSDI*.

[20] Jayaram Mudigonda et al. 2011. NetLord: A Scalable Multi-tenant Network Architecture for Virtualized Datacenters. In *SIGCOMM*.

[21] Lucian Popa et al. 2013. ElasticSwitch: Practical Work-conserving Bandwidth Guarantees for Cloud Computing. In *SIGCOMM*.

[22] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C Mogul, Yoshio Turner, and Jose Renato Santos. 2013. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 351–362.

[23] Hamed Rezaei et al. 2018. Slytherin: Dynamic, network-assisted prioritization of tail packets in datacenter networks. In *ICCCN*.

[24] Naveen Kr Sharma et al. 2018. Approximating fair queueing on reconfigurable switches. In *NSDI*.

[25] Balajee Vamanan et al. 2015. Timetrader: Exploiting latency tail to save datacenter energy for online search. In *MICRO*.

[26] Luping Wang, Wei Wang, and Bo Li. 2018. Utopia: Near-optimal coflow scheduling with isolation guarantee. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 891–899.

[27] David Zats et al. 2012. DeTail: reducing the flow completion time tail in datacenter networks. In *SIGCOMM*.

[28] Qiao Zhang et al. 2017. High-resolution Measurement of Data Center Microbursts. In *IMC*.