# Slytherin: Dynamic, Network-assisted Prioritization of Tail Packets in Datacenter Networks

Hamed Rezaei
University of Illinois at Chicago, USA
Email: hrezae2@uic.edu

Mojtaba Malekpourshahraki
University of Illinois at Chicago, USA
Email: mmalek3@uic.edu

Balajee Vamanan
University of Illinois at Chicago, USA
Email: bvamanan@uic.edu

***Abstract*—Datacenter applications demand both low latency and high throughput; while interactive applications (e.g., Web Search) demand low tail latency for their short messages due to their partition-aggregate software architecture, many data-intensive applications (e.g., Map-Reduce) require high throughput for long flows as they move vast amounts of data across the network. Recent proposals improve latency of short flows and throughput of long flows by addressing the shortcomings of existing packet scheduling and congestion control algorithms, respectively. We make the key observation that long tails in the Flow Completion Times (FCT) of short flows result from packets that suffer congestion at more than one switch along their paths in the network. Our proposal, *Slytherin*, specifically targets packets that suffered from congestion at multiple points and prioritizes them in the network. Slytherin leverages ECN mechanism which is widely used in existing datacenters to identify such tail packets and dynamically prioritizes them using existing priority queues. As compared to existing state-of-the-art packet scheduling proposals, Slytherin achieves 18.6% lower $99^{th}$ percentile flow completion times for short flows without any loss of throughput. Further, Slytherin drastically reduces $99^{th}$ percentile queue length in switches by a factor of about 2x on average.**

## I. Introduction

Datacenters have emerged as the de facto platform for hosting user facing applications that query vast amounts of Internet data (e.g., Web search) [1, 2]. To provide efficient and up-to-date access to data for these foreground applications, datacenters also run other background applications (e.g., Web crawler), which reorganize and update data. The nature of these two broad categories of applications determine the traffic dynamics and objectives of the underlying datacenter network.

Foreground applications such as Web search require access to data that is spread across a large number of servers, for each user query. Each query must wait for responses from most of the servers (e.g., 99% of servers) to achieve a good tradeoff between query response time and quality (section II). Thus, these applications generate relatively short flows (e.g., 8 - 32 KB) and are sensitive to the tail (e.g., $99^{th}$ percentile) flow completion times. On the other hand, background applications, by their very nature, generate long lasting flows which are sensitive to throughput. Therefore, a well-designed datacenter network must provide low tail flow completion times for short flows and high throughput for long flows.

Load balancing, congestion control, and packet scheduling play a key role in determining the bottomline performance of datacenter networks. Good load balancing is crucial for both latency and throughput; poor load balancing leads to congestion hotspots (i.e., long queues) that worsen (tail) latency and leads to under-utilization of network capacity (i.e., throughput loss). Fortunately, many recent proposals achieve near-perfect load balancing [3–5]. While congestion control proposals improve tail flow completion times to some extent, their main thrust lies in modulating the flow rate over several RTTs using network feedback (e.g., RTT, ECN) without causing congestion [6–10]. As such, most congestion control approaches focus on long flows which last a few tens of RTTs. Because short flows only last for a handful of RTTs, packet scheduling plays a much more direct role in determining the flow completion times of short flows. Therefore, we focus on packet scheduling in this paper.

Existing packet scheduling approaches [9–15] prioritize short flows, in an effort to mimic Shortest Job First (SJF) scheduling, which is known to minimize average flow completion times. Information-aware flow scheduling approaches [9–12] explicitly use flow sizes or deadline information to prioritize flow packets using multiple queues. Information-agnostic approaches [13] gradually demote flows from higher priority queues (i.e., every flow would start at the highest priority queue and move down in priority after sending some packets). In this paper, we ask the question: *Is it possible to further improve tail flow completion times beyond SJF scheduling?* We answer this question in the affirmative with *Slytherin*, a packet scheduling mechanism to improve tail flow completion times.

It is well known that datacenter networks experience longer tail flow completion times (e.g., 5-10x of median) due to the bursty nature of traffic [16], shallow switch buffers [6], and partition-aggregate architecture of applications, which causes incast [9, 17]. We make the *key* observation that a majority of packets that fall in the tail of the distribution incur long queuing delays at more than one switch in the network. Slytherin's key idea is to prioritize those packets that have already incurred queuing earlier in their paths. However, realizing our idea requires tackling several implementation challenges.

Our *first* challenge is to decide where to implement our scheme: in the end host or in the network. Because short flows do not offer enough time to detect and enforce priority at the end host, we opt for an in-network implementation. The *second* challenge is to identify the right network signal from which we can reliably infer queuing delays. One naive way is for

every switch to measure the waiting time in its queues and to include this waiting time in the packet header. However, this approach would require extra fields in the packet header and support for time stamping at the switches. Instead, we make a clever observation that Explicit Congestion Notification (ECN) marks already provide this information at a somewhat coarse granularity. Because ECN is readily supported and widely deployed in today's datacenters [6], relying on ECN makes our design much more implementation friendly. Inspite of the ECN's coarser granularity, we found ECN marks to work well in practice. We observe that more than 50% of tail packets incur ECN marks (incur large queuing) at more than one switch at higher loads. (section III, table I). In Slytherin, switches promotes packets that have ECN marks to higher priority queues. In contrast to existing schemes that implicitly or explicitly use static flow information (e.g., flow size, deadline) to infer priority, Slytherin infers priority based on packet queuing delays, specifically targets packets that are more likely to fall in the tail, and improves tail flow completion times beyond SJF.

In summary, Slytherin's contributions include:

- Unlike prior schemes, Slytherin specifically targets packets that are more likely to fall in the tail.
- Slytherin infers queuing delays incurred by packets without costly timestamps but relies on clever use of ECN marks.
- Slytherin *drastically* reduces $99^{th}$ percentile queue length in switches by about a factor of 2x on average.
- Slytherin achieves 18.6% lower $99^{th}$ percentile flow completion times for short flows without any loss of throughput.

The rest of the paper is organized as follows. We begin with a background of datacenter applications and overview of packet scheduling approaches in section II. We describe Slytherin's design in section III. Section IV and section V present our methodology and results, respectively. Section VI presents an overview of other related work in this area and section VII concludes our paper.

## II. Background and motivation

To motivate our design, we first address two types of flow scheduling schemes in datacenters and then we show performance trade-offs with existing scheduling proposals. We also discuss these method's ability to improve $99^{th}$ percentile[1] flow completion time of flows which suffered from queuing delay at multiple points.

### A. Information-agnostic scheduling:

As we discussed in section I, these methods try to schedule packets while there is no prior knowledge about the flow characteristics like flow size or flow deadline. In these methods, the scheduling algorithm usually assigns different priorities to packets and then each packet will be assigned to a specific queue based on the given priority. As an instance, one of past proposals, PIAS [13], tries to leverage multiple priority queues to implement Multiple Level Feedback Queue (MLFQ), in which corresponding packets of a flow get gradually demoted from higher priority queues to lower ones based on bytes it has already sent [13].

The biggest advantage of information-agnostic scheduling schemes is ease of implementation which comes from not requiring prior hard-to-achieve information about the flows (e.g., flow size). Moreover, although they may add small complexities to switching fabric but since they don't rely on prior information about the flows, they can be implemented in real datacenter networks.

The disadvantage of information-agnostic scheduling schemes is their inability to improve *tail* flow completion times. This is because, they schedule packets based on just limited information about the flow. More specifically, they try to treat each single packet (regardless of packet's history) based on a scheduling algorithm which can not discriminate among packets that suffered more queuing delay in previous hops (e.g., tail packets) and others. This is considered as a big problem because if packets of a flow experience different amount of queuing delay at multiple hops, flow completion times will be increased which is not tolerable in datacenter networks.

As a conclusion, current information-agnostic scheduling methods cannot improve tail flow completion times because of their inability to discriminate among congested packets and normal packets. In general, information-agnostic schemes provide lower performance in contrast to information-aware methods. For example, $D^3$ [10] shows that as much as 7% of flows may miss their deadlines with DCTCP [6] which is an information-agnostic scheme. However, as long as they don't require prior information about the flows, it's quite fair to not to expect them to provide as good performance as information-aware scheduling methods. We will discuss information-aware methods in the following section.

### B. Information-aware scheduling:

Information-aware scheduling schemes try to provide higher performance by giving prior information about flows to switching fabric. In the other words, these schemes promise that switches know some key information about the flows like flow deadline, flow size or even flow remaining processing time. This information is usually given to switches either by a central controller or by end host applications. In general, we can divide these methods to two different groups:

*1) Prior knowledge about flow deadlines:* These methods assume flow deadlines all are known apriori and switches greedily schedule flows with nearest deadline ahead of others. Having comprehensive knowledge about flow deadlines guarantees that almost non of deadlines are missed which drastically improves the performance of short, deadline sensitive flows. Some of methods which use this approach are $D^3$ [10], $D^2TCP$ [9], and Karuna [11].

Remember from section I that most of flows in datacenters are short and deadline sensitive. Furthermore, deadline aware

[1] we use tail FCT and $99^{th}$ percentile FCT interchangeably

schemes usually meet the requirements of vast fraction of flows in datacenters. However, although they have a big advantage over deadline-agnostic schemes but they still suffer from severe implementation issues. For many applications, such information (e.g., flow deadline) is difficult to obtain, and may even be unavailable [13].

*2) Prior knowledge about flow sizes:* These schemes assume flow sizes all are known to switches. They try to emulate Shortest Job First (SJF), which is known to minimize the average flow completion times. These methods greedily schedule shorter flows ahead of longer flows in their simplest form. In a more complicated form, they try to give higher priority to flows with shortest remaining size. Some of well known schemes that use this approach are pFabric [12], PASE [14], and PDQ [15].

Although information-aware scheduling methods provide better performance in contrast to information-agnostic approaches, but they try to use some information that are not easy to collect in datacenters. Prior proposals argue that flow sizes(or deadlines) can be tagged on packets by end host applications or a central controller may provide this information to switches. In both cases, we need to add extensive complexities to end-hosts and switches which is not only easy at all but even sometimes impossible.

As we saw, packet scheduling methods all suffer from either implementation issues or lack of smart scheduling decisions in case of scheduling congested tail packets. It turns out that most of current scheduling schemes are not effective enough or are very difficult to implement in real datacenters. Our biggest motivation is to provide a scheme which is both implementation friendly and smart enough to schedule mix flows while giving higher priorities to congested tail packets. To do so, we introduce *Slytherin* which is an information-agnostic method that not only improves tail flow completion times but even doesn't require any modifications at existing switches.

## III. Slytherin

In this section, we discuss *Slytherin* which unlike prior schemes specifically targets packets that are more likely to fall in the tail. This is important because the performance of foreground datacenter applications (e.g., Web search) are sensitive to the tail of flow completion times. Slytherin leverages Active Queue Management (AQM) schemes which are available in today commodity switches to identify congested tail packets and then assigns higher priorities to those packets.

A TCP flow is not finished successfully unless all transmitted packets reach the destination. If some packets get delayed, the total flow completion time of flow will be increased. This problem caused by delayed tail packets of a flow because they face queuing delay at multiple switches on the path. Thus, although we may have most of packets arrived in a short time but delayed tail packets will increase the total flow completion time. Our main contribution is our *novel* insight that packets that are more likely to fall the tail often incur congestion at multiple points in the network.

To achieve our goal of identifying packets that incur congestion at more than one point in the network and prioritize them quickly to improve tail flow completion times, we need:
(1) A fast and a reasonably accurate signal to pinpoint packets that are likely to fall in the tail. We set out with the explicit goals of not requiring custom hardware and supporting coexistence with legacy transport protocols like TCP [18]. To have higher response time, we should use in-network mechanisms to detect congestion at each switch independently.
(2) Provide a fast prioritization method to improve tail flow completion times. Consequently, the packet prioritization mechanism should be in-network because short flows only last for a few RTTs.

TABLE I: Slytherin's opportunity

| **Load** | 40% | 50% | 60% | 70% | 80% | 90% |
|---|---|---|---|---|---|---|
| **Fraction of packets** | 1.3% | 3.8% | 19% | 40% | 56% | 65% |

### A. Identifying tail packets

Explicit Congestion Notification (ECN) [19] is widely used AQM scheme in datacenters. With ECN, switches mark packets when the queue length exceeds a certain predefined threshold. Instead of using expensive, unreliable packet timestamps, we leverage ECN to pinpoint tail packets. More specifically, we infer that marked packets have experienced congestion in their paths, and therefore, those packets must be prioritized ahead of packets without ECN marks.

We performed an opportunity study to confirm our intuition that packets that are more likely to fall the tail often incur congestion at multiple points in the network and that we can identify those packets using ECN marks. For this study, we simulated typical datacenter traffic patterns, as reported in prior papers [16], in a spine-leaf topology with 400 hosts (see section IV). All the hosts run DCTCP. Table I shows the fraction of tail packets (i.e., here we only consider packets whose flow completion times are greater than the $95^{th}$ percentile flow completion times; these packets are more likely to impact tail FCT than packets at lower percentiles) that are ECN marked at more than one switch in their path. In other words, it shows what fraction of critical packets incur ECN marks at more than one switch. We clearly see that a *significant* fraction of these packets are marked at multiple switches. This study clearly validates our approach.

As long as each of switches perform ECN marking individually, congestion is recognized independently at each hop. This ensures that whether there is only one congestion point or multiple points of congestion, it will be reported by the corresponding switches to end hosts. Slytherin uses this bit of information not to let the end hosts to decide about the congestion but react to it just in-network. More specifically, each Slytherin switch check this bit of information individually to see whether the packet has experienced congestion at previous hops or not.

Since AQM schemes are available in today commodity switches, we can simply leverage them to discriminate among packets that suffered from congestion in previous hops vs. other

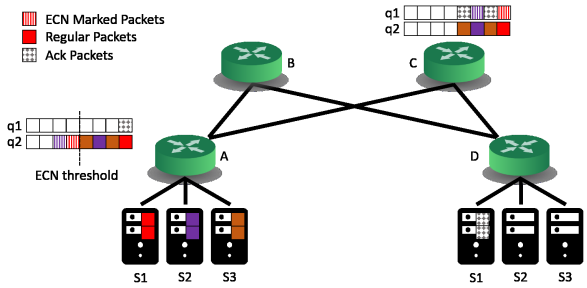



Fig. 1: Slytherin's high level idea

packets. Because ECN is meant to inform end hosts about congestion, existing switches do not check whether this bit is set or not. However, Slytherin requires switches to check this bit and prioritize such packets.

### B. Prioritizing tail packets

As we discussed above, ECN is a widely used, in-network congestion notification mechanism at switches. Our analysis shows that this bit of information should be used by the switches to prioritize previously congested packets over the others. If the CE bit in the packet's header is set, this packet is considered as a congested packet; which means it requires priority escalation at next hop switches to minimize flow latency. Scheduling packets that suffered from congestion in the higher priority queue assures that tail packets that faced congestion earlier will not be delayed at the current switch. It significantly improves tail flow completion times which is a key performance metric in datacenters.

Slytherin switches require *two* queues per switch port. Packets are assigned to one of those queues based on their priority. The first queue is a priority queue which is dedicated to congestion experienced packets and the second one is a normal queue which is shared among all other packets regardless of their flow size or flow deadline. Both queues drain packets in a First-In-First-Out (FIFO) fashion and packets in the second queue get served if and only if the first queue is empty. Figure 1 shows Slytherin's congestion detection and packet prioritization mechanism.

Alongside prioritizing congested packets, we prioritize ACK packets over other packets by scheduling them in the priority queue. Although it's not our novelty to prioritize ACK packets but our empirical analysis shows that ACK packets prioritization could be a good scheme to be used in conjunction with Slytherin.

Slytherin is designed to prioritize packets *after* the first hop upon observing ECN marks. Slytherin's Packet prioritization mechanism is totally application-agnostic. If a packet gets ECN marked, whether it's part of a short flow or a long flow, the packet will be prioritized at the next hop switch. Although prioritizing congested packets of short flows may be more efficient, it requires application knowledge and/or significant effort. Our experiments show that there is only a small degradation in performance in case of prioritizing both long and short flows in contrast to only prioritizing short flows.

**Algorithm 1** Slytherin's packet prioritization pseudocode

```
1: for Each packet "P" to be enqueued do
2:     if CE bit is set then
3:         put "P" in higher priority queue
4:     else
5:         put "P" in lower priority queue
6:     end if
7: end for
```

Algorithm 1 shows both congestion detection and packet prioritization steps at Slytherin's switches. Slytherin's design is such simple that its complete implementation requires about only 5 lines of code on the switches. Irrespective of where packets are ECN marked (i.e., enqueue vs. dequeue), Slytherin prioritizes marked packets to reduce flow completion times for both short and long flows. Overall, Slytherin's implementation is simpler than other schemes such as Pfabric[12], PIAS[13], and PDQ[15].

### C. Parameters setting

In this section, we will discuss important parameters settings that directly affect Slytherin's performance.

*1) Transport protocol:* Slytherin is designed to work in conjunction with DCTCP [6] as the underlying transport protocol. DCTCP elegantly aggregates the one-bit ECN feedback from multiple packets and multiple RTTs to form a multiple-bit, weighted-average metric for adjusting the window [9]. Using this feedback, the senders adjust their congestion window sizes in a graceful manner, so that any congestion over the path from source to destination will be proactively treated.

As long as Slytherin is an in-network scheme, it can work with any other transport protocols but we chose DCTCP due to the better throughput it provides for long flows and better flow completion time that it provides for short flows. Our analysis shows DCTCP's congestion window adjustment could be improved if ECN marked packets are drained faster at switches. In the other words, if ECN marked packets arrive faster, DCTCP adjusts its congestion window faster to avoid further congestion which leads to less packet drops, higher throughput, and lower flow completion times.

*2) ECN threshold:* Slytherin's most important parameter to tune is the ECN threshold at each switch queue. This threshold should be set carefully as it directly affects the performance. The threshold is set to 25% of the total queue size. If the threshold is too short, many packets are assigned to the higher priority queue which means still many previously congested packets will be scheduled behind others in the current switch. Briefly, if the threshold is too short, Slytherin's performance will be degraded.

If the threshold is too big, probably no packet gets prioritized over others but only some packets that are about to drop. In this case, the switch marks a lot of packets as not-congested while they are potentially congested. It means that setting up a big

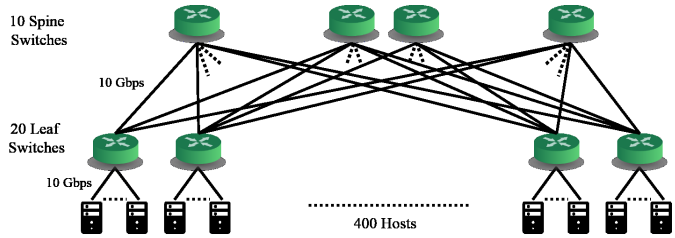



Fig. 2: Topology used in simulations

threshold leads to performance degradation as well. We will study the sensitivity of our proposed method to ECN threshold in section V.

### D. Fairness and high load scenarios

It is important to discuss Slytherin's fairness and its performance under sustained high load. Because Slytherin only prioritizes a small fraction (i.e., tail packets) of flows, fairness is largely unaffected; our results show that Slytherin achieves better tail flow completion times for short flows and better throughput for long flows than PIAS and DCTCP. Another important question is about Slytherin's performance when there is simultaneously high queuing along most (all) switches along the path. Our experiments show that it is so rare for many switches along the path to have high queuing *at the same time* that it does not affect $99^{th}$ percentile FCT (e.g., such scenarios happen less than 1% of the time).

## IV. Experimental Methodology

In this section, we present the details of our simulator implementation, topology, and workload.

*1) Topology:* We use ns-3 [20] to simulate leaf-spine datacenter topology as shown in Figure 2. Leaf-spine is a commonly used topology in modern datacenters [12]. In our simulations, the fabric interconnects 400 hosts through 20 leaf switches connected to 10 spine switches in a full mesh manner which provides over-subscription factor of 2. Each of leaf switches have 20 10 Gbps downlinks to the servers and 10 10 Gbps uplinks to the spine switches. The end-to-end Round-Trip Time (RTT) across the fabric is 80 $\mu$s.

*2) Workload and Traffic:* To evaluate our method, we simulate web search workload that is very common in modern datacenters. We consider two flow size distributions; short flows and long flows. All flows arrive according to Poisson process and the source and destination for each flow is chosen uniformly randomly. Short flow sizes are uniformly chosen in the range of 8 KB to 32 KB and our long flow's size is 1 MB. Since in web search workloads vast amount of all bytes are produced by 30% of flows that are larger than 1 MB [12], we use the same approach to produce the loads. We also use those short flows to produce incast type traffic which are quite common in web search workloads.

To further evaluate Slytherin's performance, we consider two metrics; one for short flows which is Flow Completion Time (FCT) and one for long flows which is throughput. Moreover, we check both *average and* $99^{th}$ percentile flow completion times of short flows to measure the performance of our proposed method. Similarly, we evaluate Slytherin's performance in both average and tail flow completion times in different incast scenarios. We use incast degrees of 24, 32, and 40 (number of parallel senders to a single receiver) to check the sensitivity of our method to incast degree. Next, we analyze Slytherin's performance using other metrics such as queue length, convergence speed and number of reordered packets.

*3) Compared schemes:*

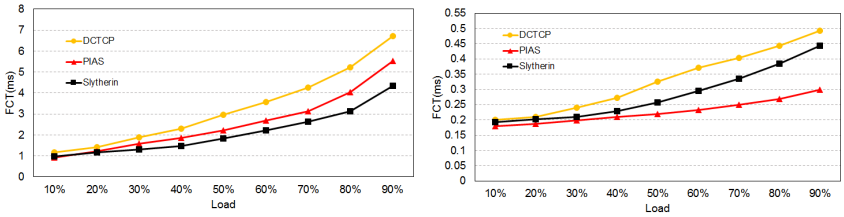
- **DCTCP:** We implemented DCTCP, capturing all details in their paper [6]. We use DCTCP as baseline.
- **PIAS:** We implemented PIAS on top of ns-3 [20] simulator. The implementation assigns 4 queues to each switch port. At the very beginning, all flows get mapped to to the highest priority queue (queue 1). If number of sent bytes of a flow reach a threshold, the priority of corresponding flow will be decreased and then the rest of packets of the flow will be assigned to lower priority queues (queues 2 to 4). We set the ECN threshold to 25% of the queue size and the load balancing method is flow ECMP.
- **Slytherin:** We implement Slytherin on top of DCTCP [6]. Our implementation uses two queues per each switch port, recall from section III that we use one high priority queue for expediting ECN marked packets and ACKs, and one low priority queue for other packets. We set the ECN threshold to 25% of the queue size and we use ECMP for load balancing. We set out the rest of our network configuration to match PIAS.
- **Ideal SJF:** We also implemented an ideal SJF scheduler. Our SJF scheduler is aware of flow sizes and maps short flows to higher priority queues. While this is not realistic as flow sizes are often not known apriori, we use our ideal SJF implementation for deeper analysis of Slytherin's queuing delays.

As such, we use the same values for parameters that are common to DCTCP, PIAS, and Slytherin (e.g., ECN threshold), and the values match those used in previous papers. There are a number of packet scheduling approaches in the last few years such as pFabric [12], PDQ [15], and PASE [14]. However, because PIAS compares to and outperforms these approaches, we only compare Slytherin to PIAS.

## V. Results

In this section, we evaluate the performance of Slytherin in different scenarios on top of ns-3 [20] simulator. Our performance evaluation consists of six parts:

- Bottomline comparison of Slytherin's Flow Completion Time (FCT) and throughput to PIAS
- Analysis of Slytherin's average queue length to explain our performance gains vs. PIAS and ideal SJF.
- Convergence analysis of DCTCP and Slytherin
- Sensitivity to varying incast degrees




(a) Tail ($99^{th}$) percentile FCT (b) Average FCT

Fig. 3: Flow completion times (short flows)

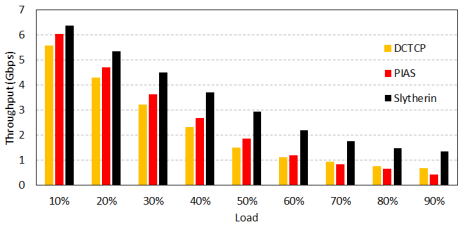



Fig. 4: Average throughput (long flows)

- Sensitivity to ECN marking threshold
- Analysis of packet reordering in Slytherin vs. PIAS.

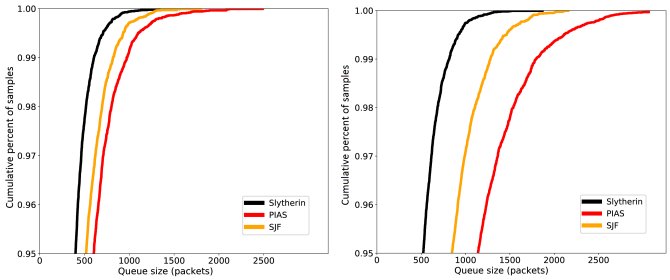
### A. Tail FCT and Throughput

In this section we will show how Slytherin performs in terms of both flow completion times (for short flows) and throughput (for long flows). The results for flow completion times and throughput are shown in figure 3 and figure 4 respectively. In both figures, the X-axis shows the load factor on network. In figure 3 the Y-axis shows flow completion time in milliseconds and in figure 4 the Y-axis shows throughput in Gbps.

*Flow Completion Time:* Figure 3a compares the $99^{th}$ percentile completion times of Slytherin and PIAS. As load increases, tail FCT increases for all the schemes. PIAS and Slytherin significantly outperform DCTCP. PIAS greedily assigns higher priority to flows that sent less packets regardless of queuing delays which leads to high queuing delay for some packets. Although PIAS benefits from ECN marking to prevent long queuing delay for longer flows, it falsely classifies some packets that have incurred higher queuing into lower priority queues which worsens tail FCT. In contrast, Slytherin targets tail packets and prioritizes them. At higher loads, Slytherin achieves better reduction in tail FCTs as there is more opportunity to schedule tail packets. Slytherin achieves an average reduction in tail FCTs of about 20% for loads greater than 20% (typical operating point of most datacenters).

Figure 3b shows the *average* FCT for short flows. Although PIAS outperforms Slytherin in average FCT but since the performance of foreground datacenter applications (e.g., Web search) are sensitive to the tail of FCT and not mean, Slytherin makes the right trade-off by prioritizing tail packets over average (or median) packets. Nevertheless, we expect to see better average FCT for PIAS because it emulates SJF, which is known to minimize average FCTs.

*Throughput:* To see the performance of Slytherin for long flows we measure *average* throughput in our simulations. Schemes that try to mimic SJF usually suffer from throughput issues; because they give strict priority to shorter flows. On the other hand, Slytherin's mechanism to expedite ECN marked packets would speed up the congestion reaction processes at end host which increases the control over sender's rate. Figure 4 shows the average throughput of Slytherin and PIAS for long flows. We see that Slytherin achieves higher throughput for long flows in contrast to PIAS. Overall, Slytherin achieves an average increase in long flows throughput by about 32%.




(a) Load=40% (b) Load=60%

Fig. 5: CDF of queue length in switches

### B. Queue length

Any packet scheduling scheme which keeps lower amount of packets in queues can successfully decrease the risk of packet drops in case of congestion. Recall from section III that since Slytherin works in conjunction with DCTCP, it tends to store lower amount of packets in queues because of its nature which transmits congested packets faster. More specifically, By expediting congested tail packets, end host's transport protocol would receive congestion signals faster and consequently it cuts its Congestion Window (CWND) faster to speed up congestion recovery processes.

In figure 5, we show the Cumulative Distribution Function (CDF) of queue lengths of all switches for Slytherin, PIAS and Ideal SJF, for 40% and 60% load. In ideal SJF, we assume that we know flow sizes apriori and switches can, therefore, faithfully implement SJF.

Figure 5b shows with a higher load factor (60%), the stored number of packets in queues increases for all schemes but Slytherin outperforms others even more. Overall, Slytherin significantly reduces $99^{th}$ percentile queue length in switches by about a factor of 2x on average. While SJF is known to substantially improve *average* flow completion times, our comparison with ideal SJF highlights Slytherin's ability to *specifically* target and improve tail latency beyond SJF.

### C. Convergence time

Expediting ECN marked packets helps TCP flows to converge faster to their fair share bandwidths when multiple

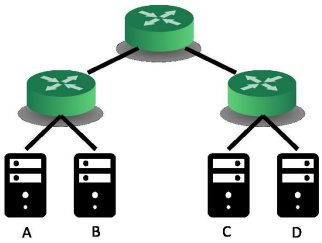



Fig. 6: Scenario used for convergence time evaluation

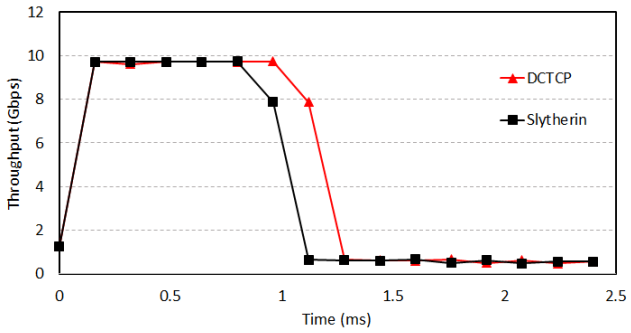



Fig. 7: Convergence time

flows compete on a bottleneck link. We evaluate Slytherin's convergence using a small network (Figure 6) consisting of four servers. We initiate a long flow from host A to host C and then, after 10 RTTs, we start 20 concurrent flows from server B to server D.

Figure 7 shows the convergence time (i.e., time to reach the fair share rate) of DCTCP (baseline) and Slytherin; X-axis shows time and Y-axis shows throughput (measured per each RTT). We see that DCTCP takes 6 RTTs (480 $\mu s$) to converge to fair share whereas Slytherin converges in 4 RTTs (320 $\mu s$). Because Slytherin provides faster convergence, it effectively mitigates utilization and fairness issues in multi-bottleneck scenarios, as reported in prior studies [21].

### D. Sensitivity to incast degree

We study Slytherin's sensitivity to incast degree and compare its tail flow completion times to those of PIAS. Figure 8 shows the $99^{th}$ percentile flow completion times of Slytherin and PIAS as we vary the incast degree as 24, 32 (our default), and 40 along X-axis, for different loads. As expected, the flow completion times increase with increasing incast degree and load. Slytherin relative gains are robust across incast degrees and loads. Overall, Slytherin achieves an average reduction in $99^{th}$ percentile FCT by about 21%.

### E. Sensitivity to ECN threshold

Next, we analyze the Slytherin's sensitivity to ECN threshold. While a lower threshold would aggressively mark packets, promote more packets to the high priority queue, and cause congestion, a higher threshold would be slow to react to congestion. Figure 9 shows the tail flow completion times of

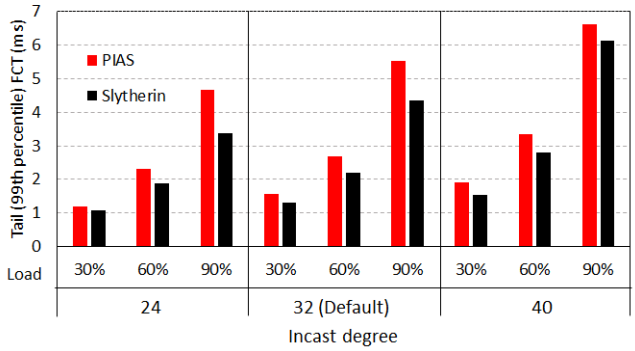



Fig. 8: Sensitivity to incast

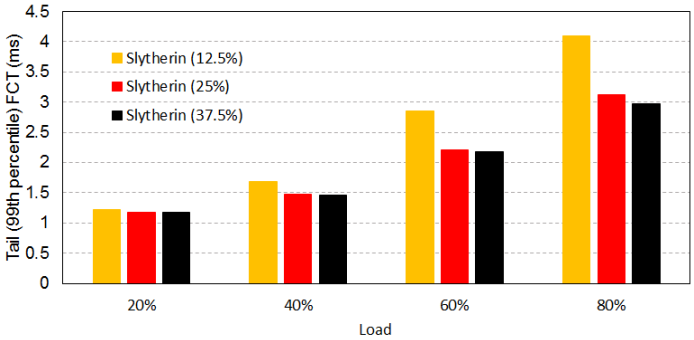



Fig. 9: Sensitivity to threshold

Slytherin for three different ECN threshold values of 12.5%, 25%, and 37.5% of total buffer size for varying loads. We see that Slytherin with threshold of 12.5% of queue szie suffers at higher loads as more packets get promoted to high priority queue and cause congestion. While a larger threshold of 37.5% of total buffer size achieves better (lower) $99^{th}$ percentile FCT at higher loads, we observed loss of throughput for long flows (not shown) due to slower reaction to congestion.

### F. Packet Reordering

In this section we investigate the effect of expediting ECN marked packets on TCP packet reordering. Slytherin schedules congested packets ahead of others to decrease tail flow completion time which may cause packet reordering at the end hosts. We compare the number of reordered packets in Slytherin and PIAS to check Slytherin's performance in terms of packet reordering efforts. PIAS authors assume the switch has enough rooms in the buffer so that each of those sub-queues (e.g., priority queues) can accommodate all incoming packets. Furthermore, PIAS sets a priority for the whole flow and consequently the number of reordered packets could be nearly zero. However, in a more realistic scenario, when the capacity of each of those sub-queues is limited, PIAS needs to either drop or demote packets to a lower priority queue which both lead to significant packet reordering.

Figure 10 shows the number of reordered packets in Slytherin compared to PIAS. X-axis shows the load factor on network and Y-axis shows the ratio of PIAS's number of reordered packets

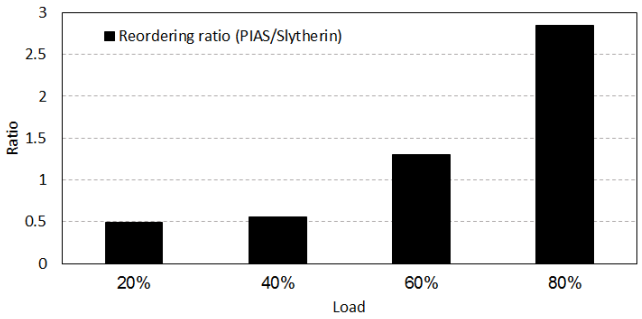



Fig. 10: Packet reordering ratio (PIAS/Slytherin)

to Slytherin. As shown in figure 10, while PIAS reorders fewer packets than Slytherin at low loads, PIAS incurs higher packet reordering than Slytherin at high loads (i.e., at 60% and higher loads). As load increases, PIAS demotes more packets. Our experiments show that while Slytherin reorders only 0.56% of total number of packets (on average), PIAS reorders 1.2% of all packets (on average) across all loads. Because the absolute number of packet reordering would be far greater at high loads than at low loads, Slytherin is more effective than PIAS in reducing reordering effort.

## VI. Related Work

There are many of past work that deal with the subjects of datacenter flow scheduling. These schemes usually require hardware modifications or rely on prior knowledge about the flows. A comprehensive review of all past proposals is beyond the scope of this paper, but we summarize some of the most relevant work here.

Earliest Deadline First (EDF) [22] is one of the oldest packet scheduling algorithms and is provably optimal when flow deadline is tagged on each single packet. D3 [10] suggests to assign different rates to flows based on their sizes and deadlines, but $D^2TCP$ [9] and MCP [23] both try to provide deadline-aware ECN-based congestion window adjustment.

Some other methods try to use prior information about flow sizes. pFabric [12] and PDQ [15] both try to schedule packets based on flow size or flow remaining size so that the shortest flow will get the higher priority. On the other hand, other schemes like [24] try to use indirect methods to assign different priorities to flows. As an instance, HULL [25] tries to improve the speed of DCTCP's congestion window adjustment by trading bandwidth. While some methods like PASE [26] use Shortest Remaining Processing Time (SRPT), other schemes like UPS [27] aim to minimize FCT by leveraging Least Slack Time First (LSTF) techniques that seem to be nearly optimal.

There are many other schemes that can not be considered as packet scheduling algorithms but they still try to minimize FCT. For example, Rate Control Protocol (RCP) [28] can achieve significant improvement in FCT of short flows. RCP is nearly optimal if minimizing FCT is the only metric for evaluation performance. RCP replaces TCPs slow start mechanism with an alternative approach that allocates fair share bandwidth to all flows at the bottleneck links. Similar to D3, RCP requires hardware modifications at switches which makes it difficult to implement. QCN [29] is a congestion control scheme that proposes a new method to improve performance in datacenters by sending congestion feedback from switches to end hosts. By utilizing intelligent switches and the new reaction logic in the end host NICs, QCN reduces recovery times during congestion; but it still suffers from implementation issues. VCP [30] is another similar scheme that relies on a mechanism like ECN feedback.

## VII. Conclusion

We presented Slytherin, which identifies tail packets and prioritizes them in the network switches. Unlike prior approaches that emulate SJF scheduling of packets which is well-known to minimize average flow completion times, Slytherin optimizes tail flow completion times, a metric that is more relevant for many online datacenter applications (e.g., Web search, Facebook). Slytherin does not require extensive support for identifying tail packets and leverages already available congestion signals (i.e., ECN). Using realistic workloads on typical datacenter network topologies, we showed that Slytherin reduces tail flow completions by about 18% as compared to existing state-of-the-art schemes. Further, we also showed that Slytherin drastically cuts the queue lengths and speeds up convergence. We plan to investigate how to further improve Slytherin's accuracy in identifying tail packets and work on efficient switch hardware implementations in the future. As data continues to grow at a rapid rate, schemes such as Slytherin that minimize network tail latency will become even more important.

## References

[1] L. A. Barroso and U. Hoelzle, *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 1st ed. Morgan and Claypool Publishers, 2009.

[2] L. A. Barroso, J. Dean, and U. Hölzle, "Web search for a planet: The google cluster architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22–28, Mar. 2003.

[3] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, "Presto: Edge-based load balancing for fast datacenter networks," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15, 2015, pp. 465–478.

[4] 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 *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14, 2014, pp. 503–514.

[5] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '16. ACM, 2016, pp. 10:1–10:12.

[6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proceedings of the ACM SIGCOMM 2010 conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 63–74.

[7] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. ACM, 2015, pp. 523–536.

[8] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, "Timely: Rtt-based congestion control for the datacenter," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. ACM, 2015, pp. 537–550.

[9] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12, 2012.

[10] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: meeting deadlines in datacenter networks," in *Proceedings of the ACM SIGCOMM 2011 conference*, ser. SIGCOMM '11. New York, NY, USA: ACM, 2011, pp. 50–61.

[11] L. Chen, K. Chen, W. Bai, and M. Alizadeh, "Scheduling mix-flows in commodity datacenters with karuna," in *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, ser. SIGCOMM '16, 2016, pp. 174–187.

[12] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 435–446.

[13] W. Bai, K. Chen, H. Wang, L. Chen, D. Han, and C. Tian, "Information-agnostic flow scheduling for commodity data centers." in *NSDI*, 2015, pp. 455–468.

[14] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar, "Friends, not foes: Synthesizing existing transport strategies for data center networks," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 491–502.

[15] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 127–138.

[16] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '10. New York, NY, USA: ACM, 2010, pp. 267–280.

[17] B. Vamanan, H. B. Sohail, J. Hasan, and T. N. Vijaykumar, "Timetrader: Exploiting latency tail to save datacenter energy for online search," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*. ACM, 2015, pp. 585–597.

[18] J. Postel, "Transmission control protocol," 1981.

[19] K. Ramakrishnan, S. Floyd, and D. Black, "The addition of explicit congestion notification (ecn) to ip," Tech. Rep., 2001.

[20] "NS-3 network simulator," http://www.nsnam.org/.

[21] I. Cho, K. Jang, and D. Han, "Credit-scheduled delay-bounded congestion control for datacenters," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 239–252.

[22] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[23] L. Chen, S. Hu, K. Chen, H. Wu, and D. H. Tsang, "Towards minimal-delay deadline-driven data center tcp," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM, 2013, p. 21.

[24] A. Munir, I. A. Qazi, Z. A. Uzmi, A. Mushtaq, S. N. Ismail, M. S. Iqbal, and B. Khan, "Minimizing flow completion times in data centers," in *INFOCOM, 2013 Proceedings IEEE*. IEEE, 2013, pp. 2157–2165.

[25] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is more: trading a little bandwidth for ultra-low latency in the data center," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 19–19.

[26] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar, "Friends, not foes: synthesizing existing transport strategies for data center networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 491–502, 2015.

[27] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker, "Universal packet scheduling," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'16, 2016, pp. 501–521.

[28] N. Dukkipati, M. Kobayashi, R. Zhang-Shen, and N. McKeown, "Processor sharing flows in the internet," in *Proceedings of the 13th International Conference on Quality of Service*, ser. IWQoS'05. Springer-Verlag, 2005, pp. 271–285.

[29] R. Pan, B. Prabhakar, and A. Laxmikantha, "Qcn: Quantized congestion notification an overview," 2007.

[30] Y. Xia, L. Subramanian, I. Stoica, and S. Kalyanaraman, "One more bit is enough," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4, pp. 37–48, 2005.