Detecting Stragglers in Programmable Data Plane

Riz Maulana
Eindhoven University of Technology
Eindhoven, The Netherlands
m.r.maulana@tue.nl

Habib Mostafaei
Eindhoven University of Technology
Eindhoven, The Netherlands
h.mostafaei@tue.nl

Nirvana Meratnia
Eindhoven University of Technology
Eindhoven, The Netherlands
n.meratnia@tue.nl

***Abstract*—Flow scheduling mechanisms in modern datacenters aim to reduce flow completion time (FCT). However, scheduling mechanisms that operate without prior knowledge, such as PIAS (NSDI 2015), or with imprecise flow information like QClimb (NSDI 2024), can inadvertently introduce *stragglers*–packets within a flow that experience significantly higher queueing delays than others. These stragglers can lead to prolonged FCT, undermining the goals of flow scheduling. While existing network monitoring tools focus on root causes of performance bottlenecks, they lack mechanisms for detecting "victims" of such issues. In this paper, we present STRAGFLOW, a data-plane tool for straggler detection. STRAGFLOW monitors queueing delays at line rate, identifies stragglers in realtime, and reports them to the control plane. We evaluate STRAGFLOW using real-world network traces and demonstrate that it can effectively detect stragglers across different scheduling schemes and various link conditions. Our results show that STRAGFLOW can provide valuable insights into straggler distribution, helping operators diagnose and mitigate flow scheduling issues to improve overall network performance.**

I. INTRODUCTION

Flow monitoring is essential to ensure reliable and efficient network operations. By monitoring critical metrics such as throughput [1], [2], latency [3], and packet loss [4] for individual flows, operators gain valuable insight into how traffic behaves. These insights are critical for maintaining service level agreements (SLAs) in datacenters and enterprise environments. For example, by monitoring flow-level latency, operators can detect when a service does not meet its latency guarantees [5], [6].

The information obtained from flow monitoring can directly inform flow scheduling decisions. An effective scheduling mechanism aims to allocate resources in a way that meets performance goals such as minimizing flow completion times (FCT) [7] or ensuring fairness among competing traffic [8]. For example, a scheduling mechanism aiming to minimize FCT can prioritize small flows based on monitored flow sizes [9]. Without up-to-date monitoring data, flow scheduling can result in inefficient resource use and missed opportunities for performance optimization.

Flow scheduling mechanisms that prioritize certain traffic classes can inadvertently create ***stragglers***—packets within a flow that experience disproportionately high queueing delays. For instance, prioritization schemes that favor small flows can cause stragglers in large flows by forcing them to endure extended waiting times. These stragglers can extend FCT and disrupt application performance. Identifying these stragglers is therefore crucial for ensuring that flow scheduling operates as intended and for detecting any misconfigurations.

ISBN 978-3-903176-72-0 © 2025 IFIP

In recent years, advances in programmable data plane technology [10], [11] have significantly enhanced flow monitoring by offering fine-grained visibility at line-rate. Unlike traditional monitoring tools such as sFlow [1] and NetFlow [2], programmable data plane can be programmed to monitor and analyze individual packets (and flows) as they pass through the data plane. This fine-grained visibility allows for the early detection of network issues like congestion [12] and microbursts [13], which are often missed by traditional coarse-grained monitoring tools.

Existing flow monitoring solutions focus primarily on identifying the root causes of network performance issues but do not address the detection of "victims" by such issues, e.g., stragglers. For example, ConQuest [12] is a data structure designed to determine which flows occupy the most space in a queue, helping operators implement active queue management. Similarly, PrintQueue [14] focuses on tracing the culprit flows that contribute to queueing delays, either directly or indirectly, using packet provenance analysis over time. While these approaches effectively pinpoint sources of congestion and delay, they do not address identifying the flows impacted by these issues. This lack of attention to victim flows leaves a gap in understanding the real impact of flow scheduling, which is crucial for making informed scheduling decisions and maintaining application-level performance guarantees.

In this paper, we introduce **STRAGFLOW**, a tool for detecting and reporting stragglers that operates entirely within the data plane. By providing immediate feedback on which packets or flows experience excessive queueing delays, STRAGFLOW enables network operators to examine the flow scheduling mechanism and take proactive measures such as priority adjustments to mitigate performance issues and reduce overall FCT. We evaluate STRAGFLOW by comparing the number of stragglers detected on CAIDA network traces based on two different scheduling mechanisms: PIAS [9] and FIFO. The results show that PIAS reduces the number of detected stragglers by up to 49% compared to FIFO. The results also demonstrate that STRAGFLOW provides valuable insights for network operators, particularly for debugging flow scheduling mechanisms.

The remainder of this paper is organized as follows. Section

2 provides background and motivation. Section 3 details the design of STRAGFLOW. Section 4 describes the evaluation, which includes the experimental setup and a discussion of the results. Section 5 discusses the related works, and Section 6 concludes the paper.

II. BACKGROUND AND MOTIVATION

A. Background

Stragglers have been extensively studied in the systems community [15], [16]. Given a job consisting of multiple tasks, a straggler refers to a task that takes disproportionately longer time to complete compared to most other tasks in the job [17]. Straggler can occur due to various reasons, including hardware heterogeneity (e.g., differences in CPU speeds, memory capacities, storage devices), resource contention, and hardware defects. Since stragglers delay the completion of the entire job, they can significantly degrade small jobs that are typically used when running interactive applications [17]. Early detection of stragglers is crucial for minimizing their impact, enabling system operators to apply techniques such as blacklisting, speculative execution [16], or cloning [17]. These techniques aim to mask slow tasks by running additional copies or reallocating resources to ensure that overall job completion times remain predictable and efficient.

Definition: Straggler. We define a straggler as a packet that experiences a higher queueing delay compared to most packets in a flow, thereby extending the time it takes for the entire flow to complete, i.e., FCT. Similarly to straggler tasks known in the systems community that can be caused by scheduling mechanisms [16], [17], we argue that network flows can also have stragglers that are caused by scheduling mechanisms. More specifically, flow scheduling mechanisms that prioritize certain flows over others inadvertently push some flows into lower priority queues where they can accumulate longer waiting times in different priority queues [7], [9], [18], [19].

B. Motivation

Flow scheduling mechanisms generally fall into two categories [20]: *clairvoyant* and *non-clairvoyant*.

Clairvoyant schedulers assume access to detailed information about flow characteristics such as flow size, enabling them to make more informed scheduling decisions. This approach works well in controlled or specialized environments, such as datacenters, where flow sizes can be accurately predicted or are inherently known, allowing the scheduler to allocate resources and prioritize flows with near-optimal efficiency. pFabric [7], pHost [21], and Homa [22] are examples of clairvoyant scheduling that attempt to reduce FCT based on prior knowledge of flow size information. However, flow information may not always be available, meaning that when a flow starts, the scheduling mechanism does not know a priori how many packets or how much data the flow will eventually contain to be sent.

Non-clairvoyant schedulers operate with limited knowledge about flow information, such as flow size. Mechanisms like PIAS [9] or QClimb [20] aim to reduce FCT without prior knowledge of exact flow sizes by dynamically adjusting priorities based on observed flow behavior. PIAS works by initially placing packets of flows in the highest priority queue. As flows continue sending packets, they can be deprioritized to lower priority queues. QClimb prioritizes network flows using imprecise size estimations rather than exact flow size knowledge. It employs machine learning techniques to estimate the lower and upper bounds of flow sizes, prioritizing the transmission of small flows over large ones based on the lower bound. The QClimb scheduler maps each flow's packets to a priority queue according to its lower bound and dynamically promotes or demotes the flow based on its remaining packet count. While this adaptability is beneficial in diverse and unpredictable network conditions, it may also lead to stragglers if certain flows are repeatedly deprioritized.

What can go wrong with non-clairvoyant schedulers? We now explain the potential problems that can arise while deploying the non-clairvoyant scheduling mechanisms.

1) PIAS and QClimb dynamically adjust scheduling based on estimated flow sizes. However, PIAS initially places all flows in high-priority queues, and if a flow grows unexpectedly, it may cause priority inversion, where large flows remain in low-priority queues. Similarly, QClimb relies on imprecise flow size predictions, leading to misclassification and inefficient scheduling. These inaccuracies can cause contention, delaying latency-sensitive applications and reducing overall scheduling efficiency. Proactively detecting and correcting such misclassifications can help mitigate long-tail latency and improve flow prioritization.
2) In priority-based scheduling, large flows may experience starvation when they are stuck in lower-priority queues, particularly under network congestion. Additionally, head-of-line (HoL) blocking occurs when a large flow remains at the front of a queue, delaying smaller flows behind it. These issues can lead to significant performance degradation, especially for delay-sensitive workloads. Detecting instances of starvation and HoL blocking enables network operators to adjust queue management strategies, ensuring fairness and maintaining predictable performance across all flows.
3) Flow scheduling mechanisms are typically designed based on general assumptions about traffic patterns. However, in multi-tenant environments or in workloads with unique characteristics–such as incast traffic, elephant flows, or deadline-driven traffic–these assumptions may not apply. When non-clairvoyant scheduling is used, it can lead to unfair resource allocation, where some tenants or specific types of flows may experience significantly increased delays. Detecting these imbalances is crucial to ensure consistent network performance and meet service level objectives.

Flow scheduling example. Let us consider a flow scheduling example using PIAS as illustrated in Figure 1 with three priority queues, i.e., Q1, Q2, and Q3. PIAS places all new

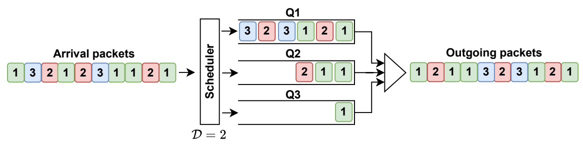


Figure 1: An example of non-clairvoyant scheduling (PIAS) with three different priority queues.

flows in the highest priority queue (i.e., Q1) and demotes them once they exceed a certain demotion threshold, i.e., $\mathcal{D} = 2$ in this example. In doing so, large flows eventually end up in the lowest priority queue (i.e., Q3).

Suppose that three flows, namely *Flow 1*, *Flow 2*, and *Flow 3*, arrive with five, three, and two packets, respectively, and the demotion threshold is set to two, i.e., only two packets per flow can stay in each higher-priority queue before being demoted. In this example, the number in each colored box in Figure 1 indicates the flow number to which the packet belongs. Considering the demotion factor of two packets for PIAS, the packets of *Flow 1* will be demoted twice since it has five packets. *Flow 2* has three packets, and one of its packets will be demoted to Q2. *Flow 3* has two packets to transmit, and they remain in the highest priority queue (Q1) for their entire duration, thereby minimizing their completion time. Running the above-mentioned flow scheduling mechanism using PIAS, the last packet in *Flow 1* resides in the lowest priority queue (Q3), experiencing higher queueing delays than the rest of the packets, making it a potential straggler.

To validate the behavior of PIAS in the previous example, we conducted an experiment to measure the queueing delay experienced by each packet in each queue. We deployed PIAS with eight queues on a Tofino switch [23] and used anonymized Internet traces from CAIDA [24] from March 17, 2016, as the workload. The trace contains approximately 30 million IPv4 packets and 1.2 million IPv6 packets, but we focus only on IPv4 packets. Each PIAS queue has a demotion threshold, determined as follows. Given a flow size distribution and N queues, the first demotion threshold is set at the $100/N$th percentile flow, the second threshold at $200/N$th percentile flow, and so on [9]. Since we use eight queues, we adopt the eighth percentile of the flow size distribution from the trace, which equals 1 packet. Based on this distribution, we set the demotion thresholds as 100, 200, 300, and so on, incrementing by 100 for queues 1 through 8, respectively. We then injected the network trace by sending the traffic from a host to the Tofino switch at 1 Gbps. The ports of Tofino switch are then throttled with two bandwidth configurations: *a balanced setting* (1 Gbps input and 1 Gbps output) and *an uncongested setting* (1 Gbps input with 1.5 Gbps output).

Figure 2 shows the distribution of the resulting queueing delay in each queue. In higher priority queues, the delays remain consistently lower, and as packets move to queues with lower priority, the average delays begin to rise. However, beyond queue 4, the observed delays flatten and stay roughly

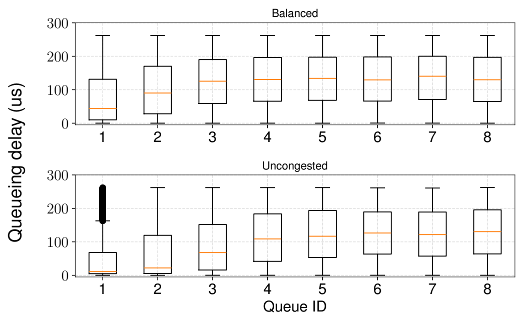


Figure 2: Queueing delay distribution of PIAS with 8 strict priority queues.

the same all the way to queue eight. This delay plateauing is caused by the flow size distribution in the network trace. Specifically, only a small subset of flows grows large enough to be demoted into those lower-priority queues, i.e., those queues see fewer packets. Additionally, because the balanced setting operates with an output rate that matches the input rate, queues are more prone to build up, yielding higher delays compared to the uncongested setting, where surplus output capacity keeps queues from becoming saturated. Overall, this result confirms that deprioritization in flow scheduling mechanisms can cause stragglers.

The need for straggler detection. The experiment results above motivate us to focus on detecting stragglers, which offer valuable diagnostic insights into the underlying scheduling dynamics in a network. By pinpointing which packets experience unexpected spikes in queueing delay and the specific queues in which these delays occur, network operators will be able to troubleshoot the scheduling mechanisms responsible for handling network traffic. The level of granular visibility provided by programmable data plane, in turn, enables them to make more informed decisions about priority configurations and flow management.

III. Design of StragFlow

We propose a tool capable of detecting stragglers in the data plane, which we call it StragFlow. By offloading the detection tasks to the data plane, we can avoid the latency and overhead associated with server-based solutions and enable faster detection. After being detected, the stragglers are reported to the control plane.

The key to detect stragglers in StragFlow is the queueing delay metadata provided by the switch hardware, `deq_timedelta`, which records how long each packet waits (in nanoseconds) in the flow scheduler's assigned queue. We leverage `deq_timedelta` to identify packets experiencing unexpected delays and classify them as stragglers. Since this metadata is only available in the egress pipeline of programmable switches, StragFlow's core functionality resides there. As packets traverse the switch, we record their `deq_timedelta`, apply our detection logic to identify stragglers, and report them to the control plane in real time.

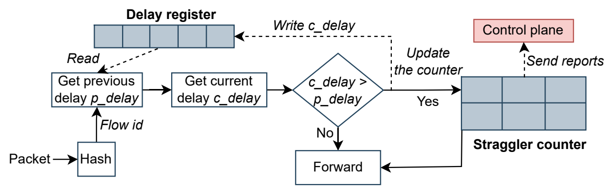


Figure 3: Overview of detection mechanism of StragFlow.

Figure 3 provides an overview of the StragFlow detection mechanism. First, an incoming packet is hashed based on its flow ID. The resulting hash value is used to locate the corresponding entry in the `Delay register`, which stores the previous queueing delay (*p_delay*). The packet's current delay (*c_delay*) is then obtained from the switch hardware's `deq_timedelta` metadata. Next, the system checks whether *c_delay* exceeds *p_delay*; if it does, the packet is flagged as a straggler, *c_delay* is written back to the `Delay register`, and the `Straggler counter` for that flow is incremented. Otherwise, the packet is forwarded immediately. Finally, any detected stragglers are reported to the control plane along with details such as the queue in which they were detected and the total number of stragglers identified so far.

In the following subsections, we explain our key design choices in detail.

A. Detecting stragglers

Determining an exact threshold for a packet's queueing delay to be qualified as a straggler can be challenging. This is because network conditions are inherently dynamic, and delays naturally fluctuate. As a result, establishing a clear and reliable threshold to differentiate normal delay variations from genuine stragglers requires careful consideration, especially in environments with multiple priority queues or mixed workloads. In the following, we present three possible ways to define a straggler concretely.

Threshold-based approach. The first approach relies on a static threshold, where the network operator manually selects a fixed delay value as the cut-off for identifying stragglers. If the queueing delay of a packet exceeds this threshold, it is marked as a straggler. Although this idea is simple, it requires careful tuning and detailed knowledge of the network traffic. On a network link with multiple priority queues, each potentially having different baseline delays, no single threshold can be universally valid. Our previous queueing delay measurements on real hardware, shown in Figure 2, illustrate that the queueing delay can exhibit an increasing trend across queues. Based on this observation, setting a static threshold may generate false positives or miss some stragglers. If the threshold is set too low, we may encounter false positives in high-priority queues, which are expected to have lower queueing delays. Conversely, if the threshold is set too high, we risk missing genuine stragglers in low-priority queues, where higher queueing delays are expected.

Moving average. The second approach is to utilize moving average to track the queueing delays experienced by a flow over time. Packets whose delays exceed the current average are then labeled as stragglers. Low-Pass Filters (LPF), available in current programmable switches such as Tofino [23], can be used to maintain and update this moving average. We can feed the current queueing delay into LPF, and the filtered result is used as a reference by the next packet to determine whether it should be classified as a straggler. This strategy can be adapted to changing workload characteristics more easily than a static threshold. However, relying on a moving average can introduce its own set of problems: (i) rapid spikes or sudden changes in delay may be smoothed out by the averaging process, causing temporary anomalies to go undetected, (ii) if traffic experiences a gradual increase in latency, the moving average may slowly drift upward, eventually making it less sensitive to emerging stragglers. As a result, the moving average method might be too reactive in some situations and too lenient in others.

Comparing with previous delay. The third approach compares the current packet's delay with the highest delay recorded for the same flow. If the current delay exceeds that, the packet is classified as a straggler, and the new maximum delay is updated. This method leverages the idea of tracking and comparing the delay history of each flow to detect stragglers based on deviations from the previously observed maximum delay.

■ ***Takeaway.*** We favor 'comparing with previous delay' method because it is straightforward to implement and can avoid the pitfalls of both threshold-based and moving average approaches: it eliminates the need for global threshold tuning and prevents delays in detecting sudden spikes due to smoothing. Each flow is effectively calibrated to its recent delay history, providing a more fine-grained and adaptive mechanism that can detect unusual increases in latency, regardless of the underlying queue configuration.

B. Counting stragglers

Considering the line-rate processing and limited memory resources in the data plane, it is essential to store flow information in a compact, collision-tolerant data structure like sketches [25], rather than maintaining a large per-flow table. Several sketches, including ElasticSketch [26], FCM-Sketch [27], and Universal Sketch [28], have been developed specifically for the use in the programmable data plane. Each of these sketches has been optimized for a different purpose. For example, ElasticSketch is designed to be adaptive to changing traffic conditions by having two separate measurement components to track large flows and small flows separately. FCM-Sketch adopts a hierarchical structure to achieve higher accuracy. Universal Sketch aims to offer a unified and general-purpose solution capable of approximating various measurement tasks—such as flow cardinality, heavy hitter detection, and entropy estimation—within a single compact data structure. However, since our goal is only to count the number of stragglers rather than to develop a new sketch, we choose the Count-Min Sketch (CMS) [29] for its simplicity and ease of implementation on a programmable data plane.

CMS strikes a balance between accuracy and memory efficiency, providing approximate counts for each flow with controlled error bounds. We use CMS to implement the `Straggler counter` shown in Figure 3. `Straggler counter` maintains a running count of the number of stragglers detected for each flow. Once we identify a packet as a straggler, we use the hashed flow ID from the previous stage to update the counters in the `Straggler counter`. Note that only straggler packets are counted; non-straggler packets are excluded from the count. Both straggler and non-straggler packets are forwarded to the designated egress port according to the forwarding rules.

C. Sending the reports to the control plane

After marking and counting the number of stragglers, we need to report the stragglers to the control plane. Tofino programmable switches support three channels for sending information to the control plane, i.e., (i) digest, (ii) CPU Ethernet port, and (iii) CPU PCIe port [30]. *Digest* is a mechanism that allows the data plane to asynchronously send summary information or specific data (such as flow IDs or queueing delays) to the control plane when certain events are triggered. The *CPU Ethernet port* is a dedicated management interface on the switch's CPU that can send Ethernet packets. This port may connect to a local CPU within the same system or to a remote control plane (e.g., an SDN controller). Finally, the *CPU PCIe port* provides a direct connection between the switch ASIC and a local CPU via PCIe. All three channels typically offer much lower bandwidth compared to the main packet forwarding bandwidth of the switch. This is because these channels are designed for control and management purposes, not for handling high-volume data plane traffic. Therefore, it is desirable to send only essential information to the control plane at a rate that the control plane can handle. To achieve this, we chose digest as the communication method between the data plane and control plane because it allows us to specify and send only the necessary information instead of transmitting entire packets. The report can include the flow ID, the straggler count, and the queue where the straggler was detected. When the control plane receives such reports and there is a significant number of them, the flow scheduler elevates that flow to the highest-priority queue for its remaining packets by installing a temporary match-action rule keyed on the flow ID. This priority boost remains in effect only while the flow continues to generate stragglers; once the straggler count falls to a negligible level, the controller revokes the rule and restores the default scheduling policy.

The naive approach is to report every single straggler to the control plane immediately, ensuring that no straggler is missed. While this approach would simplify the reporting, it would also create a flood of messages, potentially overwhelming the control plane.

To avoid flooding the control plane, our design defers reporting of stragglers until a *sending threshold* is reached. Specifically, for each flow, the data plane accumulates straggler counts in the `Straggler counter` until the count hits a preconfigured threshold. At that threshold, the data plane first sends the flow ID and the accumulated count to the control plane and then resets the counter for that flow to zero. For example, consider a sending threshold of 5 and a flow ID x with 10 stragglers. Assuming there is no hash collision in the `Straggler counter` for this flow, the data plane would send exactly two digest messages to the control plane, each containing a tuple $(x, 5)$. Since the number of stragglers is exactly twice the sending threshold, the straggler counter for flow x is reset to zero after sending the second digest message.

Trade-off of different choices regarding sending threshold: Choosing the sending threshold involves a trade-off. A higher sending threshold reduces the frequency of control plane reports, which is desirable from a scalability perspective, but increases the likelihood of missing stragglers from small flows that never reach this threshold (unless they collide in the `Straggler counter` with other flows). For instance, with a sending threshold of 5, any flow with fewer than 5 stragglers would never trigger a report, meaning that its straggler packets could go undetected. Conversely, a lower sending threshold ensures that smaller flows are not overlooked but generates more frequent reports.

■ ***Takeaway.*** Assuming sufficient bandwidth on the link between the data plane and control plane, we argue that a small sending threshold is preferable for two reasons: (i) it enables the detection of stragglers in small flows, and (ii) it allows the use of many small counters (in terms of bit size) in the `Straggler counter`, rather than a few large ones. For example, if we set the sending threshold to 10, we can use a 4-bit counter instead of an 8-bit counter, since 4 bits are sufficient to count up to 15. This approach allows for more efficient memory usage, enabling us to allocate more counters within the limited data plane memory. More small counters can help mitigate hash collisions, as the probability of two flows colliding across multiple small counters is typically lower than the probability of collision in fewer large counters. This technique can prevent excessive communication overhead while still ensuring that the control plane remains informed about stragglers.

IV. Evaluation

A. Setup

We implemented StragFlow in P4 language [31] using 740 lines of code and carried out experiments on a network testbed consisting of a Netberg Aurora 710 Tofino switch. The switch is connected to a single server that uses an Intel (R) Xeon (R) w7-3465X CPU via *Mellanox ConnectX-5* adapters responsible for generating traffic. We use two CAIDA network traces from two different dates [24], [32] as the workload. The first CAIDA trace, collected on March 17, 2016, is referred to as **C1** (the same trace as used in Section II), and the second trace, collected on January 17, 2019, is referred to as **C2**. C2 trace contains approximately 35.7 million IPv4 packets

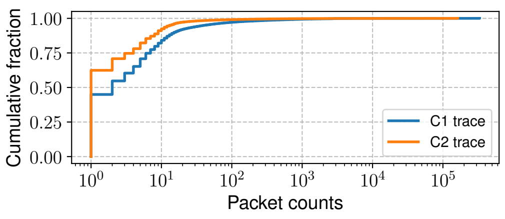


Figure 4: Flow size distribution in both CAIDA traces.

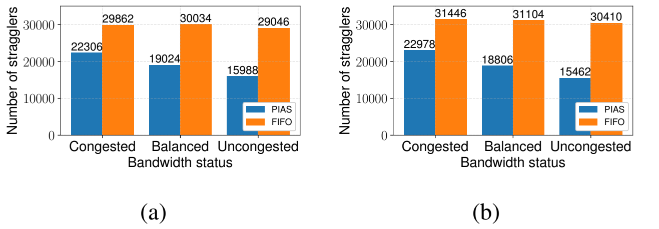


Figure 5: The number of detected stragglers on three different link scenarios when using PIAS and FIFO. a) C1 trace, b) C2 trace.

and 0.4 million IPv6 packets. Figure 4 shows the flow size distribution in C1 and C2 traces. We injected the packets of both traces using FastClick [33] into the switch at 1 Gbps speed to validate our approach.

In the evaluation, we set the sending reports to the control plane threshold to 2, as we aim to capture stragglers not only from large flows but also from small flows that may only transmit a few packets. To efficiently track the number of stragglers without excessive memory usage, we employ 65K 16-bit counters in `Straggler counter` (see Figure 3), using two hash functions to balance accuracy and space efficiency. Additionally, the `Delay register` consists of 65K 32-bit registers to store the previous queueing delay for each flow. Each bit in the `Delay register` represents one nanosecond, allowing us to record queueing delays of up to 2^{32} nanoseconds, or approximately 4.29 seconds, which is sufficient for capturing even unusually long delays.

Scenario	Ingress bandwidth	Egress bandwidth
Congested	1 Gbps	0.8 Gbps
Balanced	1 Gbps	1 Gbps
Uncongested	1 Gbps	1.2 Gbps

Table I: Network settings for various scenarios used for the experiments.

To study the impact of different network load conditions, we throttle the capacity of the ingress and egress ports of the switch connected to the server to emulate three distinct link statuses: **congested**, **balanced**, and **uncongested**. In the congested scenario, the capacity of the egress port is lower than the ingress one, creating persistent queue build-ups. For the balanced scenario, the ingress and egress ports have the same capacity, while in the uncongested case, the egress port has higher capacity than the ingress port, allowing packets to drain more quickly. Table I reports the configuration for these scenarios.

B. Straggler detection

We evaluate how STRAGFLOW can detect stragglers under FIFO scheduling with one queue and PIAS scheduling with eight queues. The demotion thresholds for each PIAS queue are set in the same way as explained earlier in Section II, i.e., we set thresholds of 100, 200, 300, and so forth for queues 1 through 8, respectively.

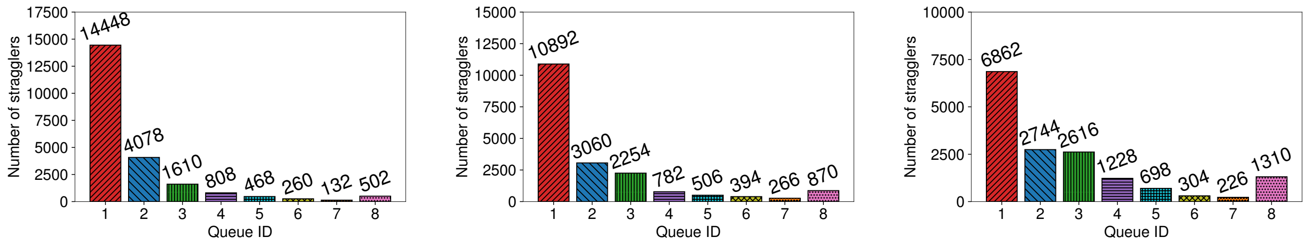
PIAS and FIFO comparison. To evaluate STRAGFLOW's effectiveness in detecting stragglers and assess the impact of different scheduling mechanisms, we compare the total number of stragglers detected using PIAS and FIFO. Figures 5a and Figure 5b present a comparison of stragglers detected in FIFO and PIAS in three scenarios with C1 trace and C2 trace, respectively.

The results of Figure 5a show that for all three scenarios PIAS reports fewer stragglers than FIFO, demonstrating that prioritizing small flows mitigates queueing delays more effectively than a FIFO queue. In the congested scenario, both PIAS and FIFO report a high number of stragglers due to insufficient egress bandwidth, leading to substantial queue buildup. However, PIAS scheduling results in 25% fewer stragglers–indicating that demoting large flows prevents them from blocking small flows. In the balanced scenario, straggler counts decrease for PIAS and slightly increase for FIFO, but the gap between PIAS and FIFO remains 37%. Finally, in the uncongested setting, the number of stragglers decreases for both mechanisms, but notable differences remain. FIFO still registers a substantial number of delayed packets, despite extra egress port capacity, indicating that queueing delays can persist even when bandwidth is available. In contrast, PIAS reduces $\approx$ 45% stragglers, demonstrating that prioritizing small flows remains beneficial even in low-congestion scenarios.

Figure 5b shows the results for the same configuration using the C2 trace. The number of stragglers in FIFO is slightly higher in all scenarios compared to C1, as the C2 trace contains more packets. The uncongested setting with C2 shows the highest reduction in stragglers, reaching about 49%.

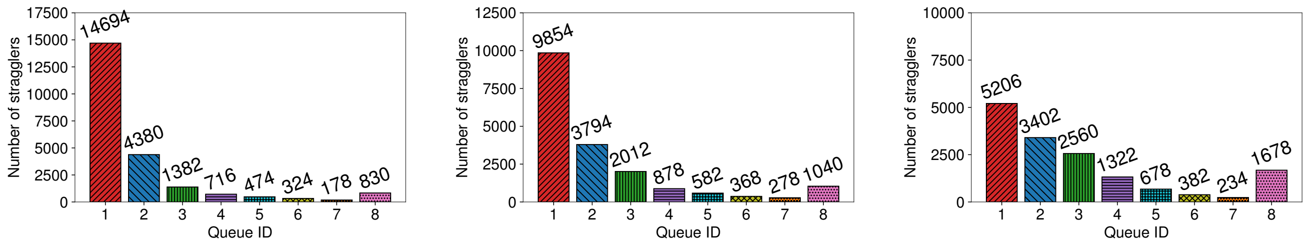
Straggler detection in PIAS. To analyze how straggler packets are distributed across different priority queues, we examine the number of detected stragglers in each PIAS queue. Since PIAS dynamically demotes flows based on their size, understanding where stragglers appear within the queue hierarchy provides valuable insights into which queues accumulate the most stragglers. Such a condition may indicate potential misconfiguration or inefficiencies in the scheduling parameters.

Figure 6 illustrates the number of straggler packets detected using PIAS in each of eight different queues when replaying C1 trace. In Figures 6a, 6b, 6c, queue 1 consistently exhibits



(a) Congested (b) Balanced (c) Uncongested

Figure 6: Detected stragglers for congested, balanced, and uncongested scenarios using PIAS (eight priority queues) on C1.



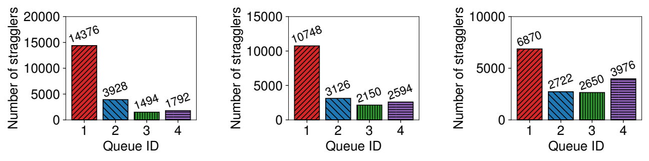
(a) Congested (b) Balanced (c) Uncongested

Figure 7: Detected stragglers for congested, balanced, and uncongested scenarios using PIAS (eight priority queues) on C2.

the highest number of stragglers, reflecting PIAS's fundamental design, where all flows start at the highest priority. Specifically, queue 1's threshold is set at 100 packets, meaning any flow with 100 or fewer packets remains in this queue for its entire duration. Analyzing the flow size distribution of C1 trace, we found that 100 packets correspond to approximately the 97^{th} percentile, indicating that nearly 97% of flows never exceed this threshold. Consequently, queue 1 experiences the highest potential for queueing delays and straggler events.

Moving beyond the first queue, the number of stragglers decreases overall, suggesting that fewer flows proceed far enough to be demoted to subsequent queues. Notably, queues 2 and 3 still see a substantial share of stragglers, indicating that mid-sized flows cross at least one or two demotion thresholds, especially under more congested conditions. By contrast, queues 4 through 8 see gradually fewer stragglers. Many of these lower-priority queues are sparsely populated, because only the largest flows (a small fraction in the CAIDA trace) continue to send enough packets to move down multiple demotion tiers.

Using the C2 trace with the same configuration, the results are similar to those obtained with the C1 trace, as shown in Figure 7. This similarity can be largely attributed to the comparable flow size distributions present in both traces. As a result, the behavior of flows under PIAS scheduling, particularly related to how flows are demoted across queues and where stragglers appear, follows similar patterns in both experiments. Therefore, the straggler distribution observed in the C2 experiment closely mirrors that of C1, highlighting the significant influence of flow size distribution on straggler behavior under PIAS.



(a) Congested (b) Balanced (c) Uncongested

Figure 8: Detected stragglers under congested, balanced, and uncongested link conditions using PIAS with four priority queues on the C1 trace.

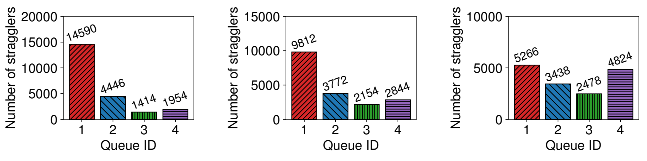
■ ***Takeaway.*** These results underscore the importance of understanding queue-level traffic patterns. If a specific queue regularly hits congestion, operators can investigate whether certain flows are being misclassified or whether the scheduling mechanism needs adjustment.

Effect of different number of queues in PIAS. We next evaluate the impact of reducing the number of priority queues in PIAS from eight to four. This experiment aims to understand how the number of queues influences straggler detection and flow scheduling behavior. With fewer queues, we expect changes in flow demotion patterns and queueing delay accumulation, potentially altering the number and distribution of straggler packets.

Figure 8 and Figure 9 report the number of stragglers detected by PIAS with four priority queues across congested,

Resource type	Match Crossbars	Gateway	Hash Bits	SRAM	TCAM	Stateful ALUs	Logical Table IDs	Number of stages
STRAGFLOW	7.29%	8.33%	7.53%	12.92%	0%	0%	16.67%	6

Table II: Resource consumption of STRAGFLOW on Tofino; all values are in percentage except the number of pipeline stages.



(a) Congested (b) Balanced (c) Uncongested

Figure 9: Detected stragglers under congested, balanced, and uncongested link conditions using PIAS with four priority queues on the C2 trace.

balanced, and uncongested scenarios using C1 and C2 traces, respectively. The straggler counts in queue 1 and queue 2 remain similar to those observed with eight queues, suggesting that most small flows either stay in higher-priority queues or experience minimal demotion. However, with fewer queues, flows that would have been more evenly distributed across lower-priority queues in the eight-queue setup are now concentrated into queue 3 and queue 4. As a result, these queues accumulate a different proportion of delayed packets, altering the distribution of stragglers.

In Figure 9, compared to the C1 trace, the results exhibit a similar overall pattern. In the uncongested setting, stragglers are more evenly distributed across all queues than in the corresponding C1 scenario. Noticeably, the number of stragglers in queue 4 is close to that in queue 1 under this setting– considering the distribution of the number of packets per flow in Figure 4. More than 75% of flows have less than 10 packets in the C2 trace.

C. Data plane resource consumption

Understanding how much resources a solution demands on real hardware is crucial to evaluate its practicality in real-world deployments. Table II summarizes the resource consumption of STRAGFLOW on the Tofino switch, indicating the percentages of match crossbars, gateway logic, hash bits, and SRAM used. Notably, STRAGFLOW requires neither TCAM nor stateful ALUs, and occupies 6 out of 12 available pipeline stages. Importantly, this utilization already includes other functionalities in the ingress pipeline, such as forwarding and PIAS queueing, rather than just STRAGFLOW alone. In general, these metrics of resource usage demonstrate that STRAGFLOW fits comfortably within the Tofino switch capabilities.

V. RELATED WORK

Network monitoring is crucial for maintaining the performance and reliability of modern networks. Various systems have been developed to provide visibility into network traffic.

Flow monitoring. Flow monitoring is a key functionality in network management to provide visibility over traffic. ChameleMon [34] introduced FermaSketch to dynamically shift measurement attention between packet accumulation and loss tasks based on the changing network state. MARS [35] is a low-cost in-band network telemetry [36] approach that periodically samples packets, employing dynamic thresholds and frequent sequence mining for self-adaptive anomaly detection. DynaMap [37] is a hybrid network telemetry system that uses dynamic query planning to map stateful operators to data plane registers at runtime according to predicted resource requirements. R-Pingmesh [38] is an end-to-end active probing system designed for RDMA over Converged Ethernet (RoCE) network monitoring, actively probing the service network to measure round trip time and end-host processing delay with minimal overhead. NetGSR [39] applies a deep learning-based technique using a generative model and a feedback mechanism to reconstruct high-fidelity network states from low-resolution measurements while adaptively adjusting the reporting rate. OmniMon [40] seeks resource efficiency and full accuracy in distributed network telemetry by combining packet embedding with out-of-band controller messages for epoch updates. Dapper [5] is a data plane performance diagnosis tool that infers TCP bottlenecks by analyzing packets in real time at the network's edge. However, existing works do not address straggler detection. STRAGFLOW fills this gap by enabling straggler identification directly in programmable switches.

Burst monitoring Burst monitoring aims to detect short-term spikes in traffic that can lead to congestion and packet loss. BurstRadar [13] detects queueing microbursts at submicrosecond resolution in programmable data planes by using a ring buffer to temporarily record telemetry data for affected packets. SpiderMon [41] also identifies microbursts and uses "wait-for" relations to pinpoint the primary flow-level contributors to the queueing delays associated with these bursts by analyzing contention among flows. NetGSR [39] demonstrates its capability to detect microbursts in ISP networks by leveraging its generative model to spot transient anomalies in traffic patterns. STRAGFLOW complements burst detection by focusing on identifying stragglers which may result from bursts.

Queue monitoring. In the networking domain, a detailed view of in-network queues is often key to diagnosing performance bottlenecks. ConQuest [12] is a data plane measurement structure that estimates flow sizes in real time and identifies the flows primarily responsible for queue occupancy. PrintQueue [14] is a monitoring system designed to track the provenance of packet-level delays over different timescales. PrintQueue categorizes the causes of queueing delay as direct, indirect, or original culprits, offering a holistic look at congestion patterns. Through time windows and a queue monitor, PrintQueue identifies which flows cause queueing delay for victim packets, going beyond simpler systems that note only the current top contributors. Although ConQuest and PrintQueue are effective in diagnosing what causes congestion or high delays, they do not address the detection of victims of such issues. STRAGFLOW focuses on detecting the stragglers in flows affected by such issues.

VI. CONCLUSION

We presented STRAGFLOW, a tool for detecting straggler packets directly in the data plane. Unlike existing flow monitoring solutions that focus primarily on identifying the root cause of congestion, STRAGFLOW shifts the focus to detecting the victims of such congestion—straggler packets that suffer excessive queueing delays. STRAGFLOW operates efficiently on programmable switches while consuming low hardware resources. The evaluation results show that STRAGFLOW effectively identifies stragglers and reveals insights into how scheduling policies such as PIAS and FIFO impact flow performance. The insights are valuable for network operators to better understand and manage the effects of flow scheduling, enabling more informed decisions to mitigate the impact of the straggler and optimize FCT.

REFERENCES

[1] "sflow," https://sflow.org/, accessed: 15-10-2024.

[2] B. Claise, "Cisco Systems NetFlow Services Export Version 9," RFC 3954, Oct. 2004. [Online]. Available: https://www.rfc-editor.org/info/rfc3954

[3] X. Chen, H. Kim, J. M. Aman, W. Chang, M. Lee, and J. Rexford, "Measuring tcp round-trip time in the data plane," ser. SPIN '20, 2020, p. 35–41.

[4] Z. Liu, S. Zhou, O. Rottenstreich, V. Braverman, and J. Rexford, *Memory-Efficient Performance Monitoring on Programmable Switches with Lean Algorithms*, ser. APoCS '20, pp. 31–44.

[5] M. Ghasemi, T. Benson, and J. Rexford, "Dapper: Data plane performance diagnosis of tcp," ser. SOSR '17, 2017, p. 61–74.

[6] S. Sengupta, H. Kim, and J. Rexford, "Continuous in-network round-trip time monitoring," ser. SIGCOMM '22, 2022, p. 473–485.

[7] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: minimal near-optimal datacenter transport," ser. SIGCOMM '13, 2013.

[8] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," *SIGCOMM Comput. Commun. Rev.*, p. 1–12, Aug. 1989.

[9] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-Agnostic flow scheduling for commodity data centers," in *NSDI '15*, Oakland, CA, May 2015, pp. 455–468.

[10] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: fast programmable match-action processing in hardware for sdn," *SIGCOMM Comput. Commun. Rev.*, p. 99–110, 2013.

[11] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends," *IEEE Access*, vol. 9, pp. 87 094–87 155, 2021.

[12] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, O. Rottenstreich, S. A. Monetti, and T.-Y. Wang, "Fine-grained queue measurement in the data plane," in *CoNEXT '19*, 2019, p. 15–29.

[13] R. Joshi, T. Qu, M. C. Chan, B. Leong, and B. T. Loo, "Burstradar: Practical real-time microburst monitoring for datacenter networks," ser. APSys '18, 2018.

[14] Y. Lei, L. Yu, V. Liu, and M. Xu, "Printqueue: Performance diagnosis via queue measurement in the data plane," ser. SIGCOMM '22, 2022, p. 516–529.

[15] S. S. Gill, X. Ouyang, and P. Garraghan, "Tails in the cloud: a survey and taxonomy of straggler management within large-scale cloud data centres," *J. Supercomput.*, vol. 76, no. 12, p. 10050–10089, Dec. 2020.

[16] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI '04*, 2004.

[17] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *NSDI '13*, Lombard, IL, Apr. 2013, pp. 185–198.

[18] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable packet scheduling at line rate," ser. SIGCOMM '16, 2016, p. 44–57.

[19] H. Mostafaei, M. Pacut, and S. Schmid, "RIFO: Pushing the efficiency of programmable packet schedulers," *IEEE Transactions on Networking*, vol. 33, no. 3, pp. 1–14, 2025.

[20] W. Li, X. He, Y. Liu, K. Li, K. Chen, Z. Ge, Z. Guan, H. Qi, S. Zhang, and G. Liu, "Flow scheduling with imprecise knowledge," in *NSDI '24*, 2024.

[21] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker, "phost: distributed near-optimal datacenter transport over commodity network fabric," ser. CoNEXT '15, 2015.

[22] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: a receiver-driven low-latency transport protocol using network priorities," ser. SIGCOMM '18, 2018, p. 221–235.

[23] "Intel intelligent fabric processors," http://bitly.ws/L5o8, 2022.

[24] "The caida ucsd anonymized internet traces [march 2016]," 2016. [Online]. Available: https://www.caida.org/catalog/datasets/passive_dataset/

[25] S. Landau-Feibish, Z. Liu, and J. Rexford, "Compact data structures for network telemetry," *ACM Comput. Surv.*, Feb. 2025.

[26] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: adaptive and fast network-wide measurements," ser. SIGCOMM '18, 2018, p. 561–575.

[27] C. H. Song, P. G. Kannan, B. K. H. Low, and M. C. Chan, "Fcm-sketch: generic network measurements with data plane support," ser. CoNEXT '20, 2020, p. 78–92.

[28] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," ser. SIGCOMM '16, 2016, p. 101–114.

[29] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *J. Algorithms*, p. 58–75, Apr. 2005.

[30] Intel, *P4-16 Intel® Tofino™ Native Architecture – Public Version*, 2021.

[31] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014.

[32] "The caida ucsd anonymized internet traces [january 2019]," 2019. [Online]. Available: https://www.caida.org/catalog/datasets/passive_dataset/

[33] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *ANCS '15*, 2015, pp. 5–16.

[34] K. Yang, Y. Wu, R. Miao, T. Yang, Z. Liu, Z. Xu, R. Qiu, Y. Zhao, H. Lv, Z. Ji, and G. Xie, "Chamelemon: Shifting measurement attention as network state changes," ser. SIGCOMM '23, 2023, p. 881–903.

[35] B. Wang, H. Chen, P. Chen, Z. He, and G. Yu, "Mars: Fault localization in programmable networking systems with low-cost in-band network telemetry," ser. ICPP '23, 2023, p. 347–357.

[36] L. Tan, W. Su, W. Zhang, J. Lv, Z. Zhang, J. Miao, X. Liu, and N. Li, "In-band network telemetry: A survey," *Computer Networks*, vol. 186, p. 107763, 2021.

[37] C. Shou, R. Bhatia, A. Gupta, R. Harrison, D. Lokshtanov, and W. Willinger, "Query planning for robust and scalable hybrid network telemetry systems," *Proc. ACM Netw.*, vol. 2, no. CoNEXT1, Mar. 2024.

[38] K. Liu, Z. Jiang, J. Zhang, S. Guo, X. Zhang, Y. Bai, Y. Dong, F. Luo, Z. Zhang, L. Wang, X. Shi, H. Xu, Y. Bai, D. Song, H. Wei, B. Li, Y. Pan, T. Pan, and T. Huang, "R-pingmesh: A service-aware roce network monitoring and diagnostic system," ser. SIGCOMM '24, 2024, p. 554–567.

[39] C. Sun, K. Xu, G. Antichi, and M. K. Marina, "Netgsr: Towards efficient and reliable network monitoring with generative super resolution," *Proc. ACM Netw.*, Nov. 2024.

[40] H. Sun, Q. Huang, P. P. C. Lee, W. Bai, F. Zhu, and Y. Bao, "Distributed network telemetry with resource efficiency and full accuracy," *IEEE/ACM Transactions on Networking*, vol. 32, no. 3, pp. 1857–1872, 2024.

[41] W. Wang, X. C. Wu, P. Tammana, A. Chen, and T. E. Ng, "Closed-loop network performance monitoring and diagnosis with SpiderMon," in *NSDI 2022*, Renton, WA, Apr. 2022.