Enabling Silent Telemetry Data Transmission with InvisiFlow

Yinda Zhang, *University of Pennsylvania;* Liangcheng Yu, *University of Pennsylvania and Microsoft Research;* Gianni Antichi, *Politecnico di Milano and Queen Mary University of London;* Ran Ben Basat, *University College London;* Vincent Liu, *University of Pennsylvania*

https://www.usenix.org/conference/nsdi25/presentation/zhang-yinda

This paper is included in the Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation.

April 28–30, 2025 • Philadelphia, PA, USA

978-1-939133-46-5

Open access to the Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

Enabling Silent Telemetry Data Transmission with InvisiFlow

Yinda Zhang[†], Liangcheng Yu[†⟆], Gianni Antichi[§⋆], Ran Ben Basat[‡], Vincent Liu[†]
[†]*University of Pennsylvania,* [⟆]*Microsoft Research,* [⋆]*Queen Mary University of London,*
[§]*Politecnico di Milano,* [‡]*University College London*

Abstract

Network applications from traffic engineering to path tracing often rely on the ability to transmit fine-grained telemetry data from network devices to a set of collectors. Unfortunately, prior work has observed—and we validate—that existing transmission methods for such data can result in significant overhead to user traffic and/or loss of telemetry data, particularly when the network is heavily loaded.

In this paper, we introduce InvisiFlow, a novel communication substrate to collect network telemetry data, silently. In contrast to previous systems that always push telemetry packets to collectors based on the shortest path, InvisiFlow dynamically seeks out spare network capacity by leveraging opportunistic sending and congestion gradients, thus minimizing both the loss rate of telemetry data and overheads on user traffic. In a FatTree topology, InvisiFlow can achieve near-zero loss rate even under high-load scenarios (around 33.8× lower loss compared to the state-of-the-art transmission methods used by systems like Everflow and Planck).

1 Introduction

Network management and monitoring tasks often rely on fine-grained telemetry data to empower network administrators and systems with insight into current conditions [1–3]. For instance, when monitoring for failures and other network anomalies, switch telemetry enables the detection of unusual traffic patterns, errors, or drops [3–5]. Similarly, when optimizing or provisioning the network, administrators can analyze trends in traffic volume, bandwidth usage, and port utilization to make decisions [6–9].

Recent research has provided compelling arguments in favor of expanding the scope and diversity of collected telemetry data. Rationales vary. Some researchers note a reduction in the timescales of congestion events and its impact on the granularity required to observe them [10–12]. Others propose enhanced measurement capabilities enabled by modern network devices that can capture buffer statistics, sketches, per-QoS counters, and more [13–16]. Still, others advocate for the advantages of per-packet in-band metrics [17–19], and some suggest increasing telemetry to strive for the highest levels of availability and Service Level Objectives (SLOs) [20–22].

Regardless of the reason, high volumes of telemetry data produced at switches are typically collected at machines placed throughout the network that are responsible for processing, collating, analyzing, and storing the gathered data [1,3–5]. The traditional method used to send telemetry reports (*e.g.*, in [23]) routes them as standard data packets that can impact user traffic. For example, in §2, we observe increases in user traffic flow completion time of up to ≈19%, which affects latency-sensitive workloads. Techniques like In-band Network Telemetry (INT) [18], In-situ Flow Information Telemetry [24], and Broadcom's IFA [25] that embed data directly into user packets impose similar overheads [19].

Acknowledging this issue, recent work has tried to reduce overheads on user traffic by sacrificing the completeness of the telemetry data. One approach is exemplified by systems like [19, 26] that reduce telemetry's frequency and/or volume through approximation, saving some bandwidth but at the cost of information loss. When tracing paths of flows, for instance, PINT [19] will miss path changes for small flows (large flows as well under flowlet routing). Even when incomplete/approximate data is sufficient, the data is still eventually transferred across the network, incurring the associated overheads of those transmissions. Another approach is taken by systems like [21, 27–30] that de-prioritize telemetry traffic in favor of user traffic. This solution, however, risks starvation and telemetry loss when links are shared. The loss can affect the accuracy/utility of monitoring applications that rely on the data. For instance, our benchmark of a path tracing utility missed around 11% of paths under this strategy (§2).

In this paper, we aim to design a novel communication substrate for telemetry data that provides both (a) high sustainable throughput (*i.e.*, throughput given the constraint of zero loss) and (b) little-to-no impact on user traffic. We present InvisiFlow, a new mechanism to isolate telemetry data from user traffic. InvisiFlow is agnostic to both the content of the telemetry data and the network's topology; consequently, it can be used as a drop-in replacement for many existing monitoring systems.

The main insight behind our work is that exploring unused network bandwidth using *congestion gradients* maximizes sustainable throughput and minimizes the external impact of telemetry traffic—similar in spirit to water always flowing toward a lower elevation, telemetry packets are sent along the gradient to neighbors with lower congestion and more space in their telemetry buffers. To tackle challenges in applying existing theory on congestion gradients (§3.2) to practical environments, InvisiFlow constructs a pull-based transmission channel in which network switches frequently send low-priority pull requests to their physical neighbors that quantify the amount of telemetry congestion at each node, and utilizes shared pipelines in switches to support the late-binding of egress ports based on telemetry buffer usage.

We implement a hardware prototype[1] of InvisiFlow on a commodity switching ASIC with P4 programmability. With testbed experiments on a leaf-spine topology, we demonstrate that InvisiFlow can efficiently explore available network bandwidth to simultaneously prevent impact on user traffic *and* loss of telemetry data, even under high load. In contrast, baselines based on the communication protocols of state-of-the-art telemetry systems exhibit drop rates of >80% when prioritizing user traffic on the same workload. Simulation results on larger networks and a range of workloads, network applications, and topologies show that InvisiFlow remains resilient and effective across a wide range of scenarios. Specifically, InvisiFlow can improve the accuracy of prior systems in per-flow accounting approximation (*i.e.*, ApproSync [31] with Count-Min sketches [32]) up to 22.5×.

2 Background and Motivation

We begin by briefly summarizing existing telemetry data transmission systems and their limitations.

2.1 Telemetry Data Transmission

Network operators design and deploy systems to collect data from the network, including host/switch counters, flow records, and log information. While some systems ingest and exploit such data locally [8, 33], most of the time, operators need to collect the data at centralized databases/repositories for subsequent processing, storage, and analysis to provide a comprehensive view of the network [34, 35].

These modern telemetry systems—regardless of their target data—share a high-level workflow, depicted in Figure 1. The systems comprise three major components. The *source* can reside on any network device (*e.g.*, switches, routers, or end hosts), encompass any target data (*e.g.*, SNMP port utilization counters, drop notifications, or NetFlow records), and may be generated at any interval, periodic or event-based. A set of *collector sinks* receive, process, and store this telemetry. Monetary incentives typically imply that the number of collectors is small compared to the number of devices (*e.g.*, < 1% as shown in Everflow [28]) and that they must aggregate telemetry data from multiple devices.

Between the above two components is a *transmission channel*, responsible for forwarding telemetry packets from sources to collectors (which is the focus of this paper). We note several requirements for this transmission channel:

(R1) High/full sustainable throughput for telemetry data. As the mechanism responsible for delivering telemetry data, network applications depend on this channel to provide high throughput with minimal loss of data. Finer-grained telemetry data allows network operators to make more informed decisions [10] and provide better services to end users; data loss can result in incomplete or inaccurate insights [30].

[1] Code is available at https://github.com/eniac/InvisiFlow.

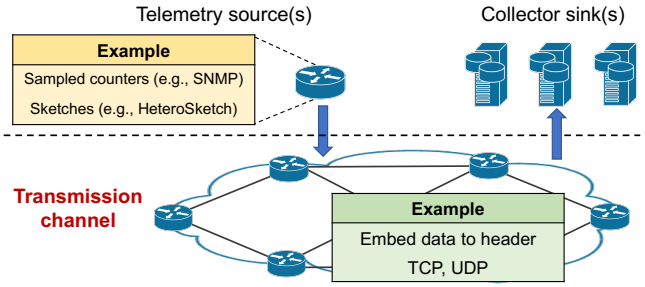


Figure 1: Workflow of a typical telemetry system.

(R2) Minimal impact on user traffic. Telemetry data often shares the same network with user traffic [19, 26, 29] as this can lead to more efficient resource utilization and beneficial fate-sharing properties [21]. While doing so, it is crucial to avoid introducing excessive latency, packet loss, or congestion due to the telemetry data transmission process, which may degrade the quality of service for user applications. As discussed in [19], this overhead can lead to up to 20% degradation of application-level throughput and a corresponding 25% increase in flow completion time.

(R3) ASIC-implemented operation. Finally, while early switches and routers often punted all non-forwarding tasks (including telemetry) to the switch CPU, growing telemetry granularity and ASIC capabilities have led to heavier reliance on data planes to both generate and send telemetry data [19, 27–30]. In particular, data planes can collect and export data at much higher resolution and power efficiency than CPUs. This trend toward data plane implementation is visible in line-rate monitoring systems like INT [18] and Broadcom's IFA [25], but it is also increasingly true for classic, decades-old telemetry systems like NetFlow and sFlow [36, 37].

2.2 Existing Solutions and Limitations

We note that recent work [14, 15, 19, 26, 38, 39] that seeks to reduce the size of collected data at the telemetry source (encompassing techniques like sampling, sketching, and encoding) can mitigate some of the above challenges; however, they are not a panacea. First, these solutions often sacrifice generality for the required bandwidth, limiting their applicability to specific applications and types of telemetry tasks. The approximation can also introduce significant errors in higher-layer analysis [5]. For example, applying PINT [19] to path tracing and the Hadoop workload [40] of §6 can limit bandwidth overheads to <2% compared to a configuration without telemetry but will miss path records for >60% of flows that are too small to track. Finally, regardless of whether the techniques are applicable/acceptable, enhancing the underlying transmission channel can mitigate the degradation in the accuracy of higher-layer applications and its impact on user traffic.

We, therefore, focus on the orthogonal problem of telemetry transmission. In this subsection, we examine the channels used by existing systems in the context of the above requirements. Per R3, we look only at data-plane solutions, of which current systems take one of the following two approaches.

Tx Method	Examples
Default-priority UDP	NetFlow [36], sFlow [42], INT-XD (Postcard) [29]
Low-priority UDP	Everflow [28], Planck [27]

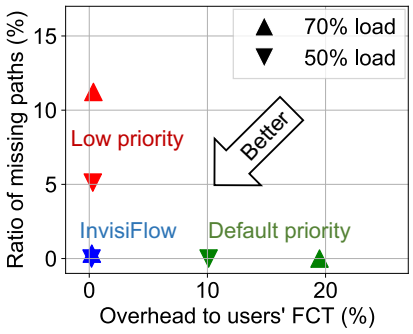


Figure 2: Tradeoffs between different prioritization methods for the telemetry transmission channel under the path tracing application. FCT dilation is an average over applications in the same rack as a collector.

Embedding data into packets. One way to transmit telemetry data is to embed data directly into user packets [18, 19, 26, 41]; end-host receivers are responsible for extracting the telemetry data from these packets and sending it to collectors. Unfortunately, as others have noted and we discussed in R2, embedded data can introduce substantial overheads [19]. For example, for small packets, adding additional data to packets can increase transmission times by a large relative proportion, increasing load on the network. For large packets, reserving header bits or reducing MTU sizes to prevent fragmentation incurs a tax on every packet, regardless of whether real data is embedded or not.

Generating packets and sending them directly. The other approach is to generate dedicated packets that are sent to collectors directly [27–30, 36, 42, 43]. These systems typically layer on top of existing protocols like UDP, which is preferable to TCP due to the impracticality of implementing end-to-end reliability in the data plane [44]. By sending telemetry data as datagrams separate from user packets, these systems can better control the rate and overhead of telemetry data transmission.

However, managing the tradeoff between the throughput of the transmission channel (R1) and its impact on user traffic (R2) can be challenging. Aside from generating less telemetry data, the primary knob to tune the overheads of today's transmission channels is Class-of-Service packet prioritization, where there are two primary options: (a) treat telemetry and user packets equally or (b) send telemetry data with a lower priority than user traffic.

Figure 2 categorizes a few popular telemetry systems along this axis and illustrates the challenge of balancing sustainable telemetry throughput and user-traffic overhead (measured via FCT dilation). Results are presented for a FatTree data center [45] using a Meta Hadoop workload [40] and a range of average utilizations, as detailed in §6.1. We employ a path tracing application similar to [5], where each switch maintains a record of user flows and forwards it to a collector for path reconstruction. As seen in Figure 2, as load increases, network operators are forced to choose between prioritizing telemetry (and increasing the FCTs of collector-collocated applications by 19%) or prioritizing user traffic (and accepting 11% missing paths). As a preview of our results, InvisiFlow can keep both minimal, even in a heavily loaded network.

3 Design Overview

In this paper, we present InvisiFlow, a communication substrate for telemetry data transmission that can minimize the network-wide loss rate of the telemetry data given the constraint of little-to-no impact on user traffic. As previously mentioned, InvisiFlow is agnostic to the content of the telemetry data and the network's topology. It also leaves user traffic transmission untouched. As such, it can be used as a drop-in replacement for many existing monitoring systems.

3.1 Opportunistic Gradient Forwarding

InvisiFlow is driven by two overarching design principles:

1. Prioritize user traffic.
2. Instead, seek out spare capacity, wherever it may be.

For the first principle and to satisfy R2, InvisiFlow keeps telemetry and user traffic separate and ensures that all control and telemetry packets are de-prioritized compared to user packets. Although packet processing and transmission are generally not preemptable in modern network devices, prior work [21, 27] and our results in §6.3 show that proper configuration of priorities at every stage of modern network devices can ensure that overheads are negligible, even when every available gap between user packets is filled with lower-priority packets. With these techniques, InvisiFlow can operate with minimal impact on user traffic.

Instead, InvisiFlow seeks out spare network capacity for telemetry data transmission, aggressively leveraging any such opportunities. Over a decade of measurement studies on large-scale networks have frequently observed low average utilization in modern networks [10, 40, 46–48], with data centers, for instance, sometimes seeing ∼5% average utilization (comprised mostly of small bursts), even for supposedly high-throughput applications like Hadoop and distributed ML training. Telemetry, generally off the critical path of user-facing requests, is a particularly good fit for leveraging the opportunities provided by these gaps in sending.

InvisiFlow's approach to finding spare capacity is based on classic theoretical foundations regarding multicommodity flows and the stability of queuing networks [49, 50]. More formally, given a network and a vector of flows with associated sources, sinks, and arrival rates, a scheduling/routing policy π is said to provide buffer stability if it satisfies the following.

Definition 1 (Buffer Stability). *Let $B(t)$ be the total amount of packets remaining in the buffers at time t. Assume no*

constraints on buffer size. The system is stable under π iff:

$$\limsup_{t\to\infty} \frac{1}{t} \sum_{\tau=0}^{t-1} \mathbb{E}[B(\tau)] < \infty \quad (1)$$

The alternative to buffer stability is a system where buffer sizes grow to infinity. Note that it is often useful to define scheduling policies that are robust to changes in arrival rates. Prior work has termed this a problem of characterizing and maximizing a policy's stability region.

Definition 2 (Stability Region)**.** *Let g be the vector of the packet arrival rates at all telemetry sources. Stability region S_π of policy π is the set of all g for which the system is stable under π.*

Results in *max-weight scheduling theory* [50] have shown that transmission based on the *congestion gradient* can provide an optimal policy that maximizes the stability region and sustainable throughput. The congestion gradient is defined as the difference in telemetry buffer usage between neighboring switches. Unlike traditional forwarding [51], where nodes blindly push packets along the shortest path to the destination, for every pair of neighbors in congestion gradient-based transmission, the one with a larger telemetry buffer usage sends toward the one with a smaller buffer usage. This is similar in spirit to water always flowing toward a lower elevation. These approaches have been found to be robust to diverse topologies and dynamic arrival patterns. As with prioritization, we argue that telemetry, typically amenable to asynchronous and out-of-order delivery, is a particularly good fit for our approach.

3.2 InvisiFlow Architecture

While InvisiFlow is inspired by the simplicity and efficacy of prioritization and congestion gradient forwarding, the theory is, of course, significantly different from practice.

(C1) How do we calculate the gradient over time and address situations where the gradient itself is not known (*e.g.*, because control packets are subject to prioritization behind user traffic)? (C2) How do we address limitations in packet processing pipeline structures for sending telemetry data, *e.g.*, the need to specify an output port before egress availability is known? (C3) How do we handle failures of both network components and telemetry collectors? The primary challenge of InvisiFlow is to develop practical solutions to the above issues and others.

Components. The high-level architecture of InvisiFlow is shown in Figure 3. It contains several types of components:

- *Switches:* To obtain the gradient (C1), every switch in the network, when it has free capacity, will continually generate low-priority pull requests (with its current telemetry buffer usage) and send them to all neighboring switches. Neighboring switches will respond to this per-hop pull request if the other side's buffer usage is lower. The current switch may also send packets if it would otherwise have dropped data. To enable the late binding of egress ports for telemetry data (C2), we store telemetry data in the egress pipeline, which is shared by multiple egress ports, as detailed in §4.1.
- *Collectors:* Telemetry collectors operate similarly to today, ingesting and processing data as it arrives. In InvisiFlow, they also act as true sinks in the congestion gradient, equivalent to having a constant telemetry buffer utilization of zero. This can be implemented by either the collector sending pull requests on its own or via offload to its last-hop switch [43]. In either case, telemetry data is naturally drawn in from surrounding devices. InvisiFlow supports sharding and backups of these devices and can support flow control through an extension of the switch protocol.
- *Available Servers:* Finally, InvisiFlow supports optional usage of spare server memory capacity, similar to [52, 53]. These also extend the per-hop pull protocol of the switches, with their advertised utilization providing a knob for tuning their usage. For example, advertising 80% usage means that they will only be used when nearby switches are >80% full. Configuring these servers is optional but can be beneficial when there is a high load or failures of either network components or telemetry collectors (C3).

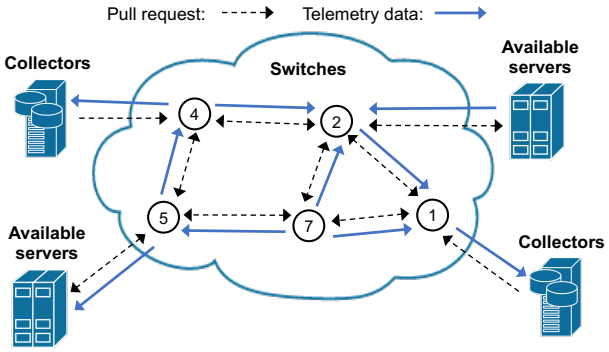
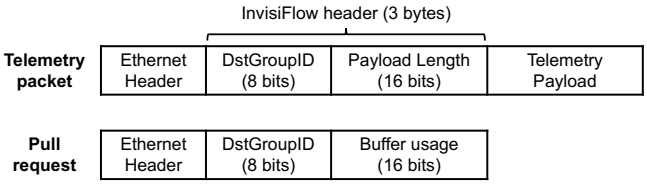


Figure 3: Overview of an InvisiFlow-enabled network.

Workflow. InvisiFlow creates a network where switches continually trade pull requests and telemetry data with their neighbors in the gaps between microbursts of user traffic. Through these gaps, InvisiFlow implements a variant of the edge-balancing approach to approximating a maximal-stability-region routing policy. As long as gaps exist, InvisiFlow will leverage them to route telemetry packets. In the rare case that one or both directions of a link are occupied for an extended period, InvisiFlow will leverage congestion gradients to find alternate paths and take risks when appropriate.

While inevitably not a perfect reproduction of the formalisms of results like [50] (*e.g.*, due to the asynchrony of pull requests), our extensive evaluation over a wide range of topologies and workloads in §6 demonstrates that InvisiFlow can—entirely in the data plane (R3)—maximize the sustainable throughput of telemetry data (R1) while incurring little-to-no impact on user traffic (R2).



Ethernet header: *Ethertype* identifies the InvisiFlow header.
DstGroupID: The destinations (collectors) group ID for the packet.
Payload length: The total length of the telemetry payload measured in bytes.
Buffer usage: The number of telemetry packets recorded in the telemetry buffer.
Telemetry payload: The telemetry data provided.

Figure 4: The packet format of InvisiFlow

Ethertype	Egress port
Generated seed packet	Multicast to all neighbors
Pull request	Ingress port
Telemetry packet	Free ports

Table 1: Default routing table used by the ingress pipeline.

4 The InvisiFlow Transmission Channel

In this section, we discuss the design of the InvisiFlow transmission channel in detail. To keep our discussion general, we assume (a) telemetry sources can be placed at arbitrary locations in the data plane and generate arbitrary amounts of data, and (b) one or more collectors are placed in the network, any of which can ingest that data. Note that InvisiFlow can also support differentiated collectors and sharding (*e.g.*, based on hashing over a partition key) through a `DstGroupID` header field and a straightforward extension to parallel transmission channels, but we omit that discussion for simplicity.

Message format. The packet formats of InvisiFlow messages are illustrated in Figure 4. There are two types of packets, both gated by `DstGroupID` as mentioned above. Pull requests are generated periodically and contain the current telemetry buffer usage, which can be scaled to account for heterogeneous buffer allocations. Telemetry data packets are only sent in response to received pull requests or upstream telemetry, and they contain a variable-length payload. Both types of packets are set to the lowest priority at every stage of the pipeline, including the ingress arbiter, traffic manager, and egress arbiter, per [21].

4.1 Switch Operation

The routing table for InvisiFlow's ingress pipeline and the pseudocode for egress pipeline processing logic are provided in Table 1 and Algorithm 1, respectively.

An important building block of this logic is the gap-filling low-priority packet generation mechanism introduced by the OrbWeaver framework [21]. OrbWeaver uses the existing per-pipeline packet generator capabilities of modern switches to generate low-priority packets sent to all ports with configurable frequency and payload, with negligible impact on the average power draw of the switch (<2%) and minimal worst-case impact on pipeline bandwidth (<1.5%). Specifically, it periodically generates 'seed' packets in the ingress pipeline

Algorithm 1: Egress logic for InvisiFlow packets.

```
Input: Packet P
switch Ethertype do
    case generated seed packet do
        Ethertype ← 'pull request';
        Set the buffer usage field;
    case pull request do
        if local.buffer_usage > P.buffer_usage then
            Ethertype ← 'telemetry data';
            Pop telemetry from buffer and append it to P;
        else
            Set the buffer usage field;
    case telemetry packet do
        Store the payload in the local buffer;
        Drop the packet;
```

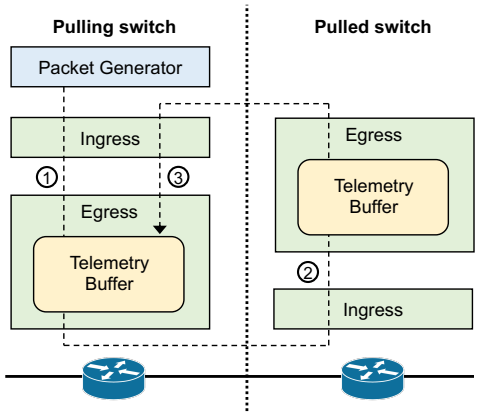


Figure 5: A potential lifecycle of a InvisiFlow packet. Circled numbers correspond to the steps in §4.1.

and then multicasts those seed packets to all egress ports, with additional details in §5 and Appendix A. InvisiFlow uses this mechanism to initiate pull requests, from which the rest of the InvisiFlow protocol stems.

Another building block is InvisiFlow's telemetry buffers in the ingress and egress pipeline of every switch. These telemetry buffers are distinct from the traditional packet buffers used by user traffic and managed by the ASIC; instead, SRAM arrays are easier for data plane programs to manipulate. Note that while both types of buffers are used to store telemetry data, the egress buffer(s) of the pipeline(s) closest to the collector(s) serves as the primary storage location. To see why, consider a system that creates and appends telemetry data to packets in the ingress. Packets must be assigned an egress port in the ingress pipeline, but the fine-grained usage of any given egress port can be difficult to ascertain, especially if multiple pipelines may be sending to the same egress port. Storing data in the egress instead enables late binding—whenever the egress arbiter can schedule a low-priority telemetry packet (destined for any port in the same pipeline), the next chunk of data can be appended immediately before it is placed on the wire. The ingress buffers are only used as a temporary

holding area for data that will eventually be transferred to the egress pipeline.

Given these components, Figure 5 shows one potential lifecycle of an InvisiFlow packet. For this packet:

1. *The pulling switch generates/sends pull requests:* As shown in Table 1, every switch in the network will periodically generate seed packets and multicast them to all neighbors. Before each copy leaves the egress pipeline, it will be converted to a pull request with the current telemetry buffer usage as shown in line 2-4 of Algorithm 1.
2. *The pulled switch replies to the pull requests:* As shown in Table 1, after receiving a pull request, the neighbor switch sets the egress port equal to the ingress port to return telemetry data. In the egress pipeline, the switch compares the telemetry buffer size recorded in the pull request with its local buffer size as shown in line 5-10 of Algorithm 1. If the local buffer size is larger, the switch pops telemetry data from the buffer and sends it back. Otherwise, the switch reflects the pull request back to the sender as its own.
3. *The pulling switch processes the telemetry data:* There are two possible replies from the pulled switch: (1) If the reply is still a pull request, the procedure remains the same as described above. (2) If the reply is a telemetry packet, we store the packet in the telemetry buffer at the egress pipeline as shown in line 11-13 of Algorithm 1.

Example. Suppose a seed packet, denoted as p, is initially generated by $S1$ and multicast to all neighbors in the ingress pipeline. In the egress pipeline of $S1$, p is converted to a pull request, p' with its buffer usage field set to the current telemetry buffer usage (*e.g.*, 10). Upon receiving the pull packet p', the neighbor switch, $S2$, routes it back out the same port. In the egress pipeline of $S2$, if the buffer usage (*e.g.*, 20) of $S2$ is larger than the buffer usage field in the pull request p', $S2$ pops telemetry data from the buffer and sends it back to $S1$. Finally, $S1$ forwards the telemetry packet through free ports and stores it in the buffer at the egress pipeline.

There are several important aspects and possible variations of the above procedure.

Egress port choice and its impact on packet buffers. First, we note that the choice of egress port of Table 1 and their configurations in each branch of Algorithm 1 is deliberate. Pull requests are valid for a single port and are reflected back for egress handling. Because the seed and pulls are regenerated periodically, the loss of these requests is acceptable, and the ASIC-level shared buffers for these packets can be shallow (*i.e.*, enough for only a handful of outstanding requests). Packets containing telemetry data should not be dropped as readily, so InvisiFlow only sends them if capacity is guaranteed. It does so by using free ports (*e.g.*, recirculation ports) that are not typically used by user traffic, as shown in Table 1. The advantage of using these ports is that it leverages higher aggregate bandwidth within the switch and reduces the usage of non-evictable packet buffers in the traffic manager.

Adding a slope to prevent zero congestion gradient. When the network contains only a limited number of telemetry packets, there are potential issues where the congestion gradient remains consistently at zero. Consider the scenario with only one telemetry packet, p, sitting in switch $S1$. A neighbor, $S2$, will pull p because it has lower telemetry buffer usage; however, $S1$ now has less telemetry buffer usage than $S2$, so there is a chance that p can oscillate between these switches.

To address the problem, we add a 'slope' to the congestion gradient. Suppose the buffer usage of switch $S1$ is u_{S1}. Switch $S1$ can pull telemetry data from switch $S2$ if and only if $u_{S1} + \delta_{S1,S2} < u_{S2}$, where $\delta_{S1,S2}$ indicates the slope. In InvisiFlow, we set δ based on the distance to the collector. If switch $S1$ is closer to the collector than switch $S2$ as defined by the routing protocol, then $\delta_{S1,S2} = 0$. Otherwise, $\delta_{S1,S2} = 1$. This approach improves the directness of paths and ensures that even when there is just one telemetry packet, it can efficiently flow toward the collector without getting stuck.

Adjust the pulling rate based on local buffer usage. Adjusting the pulling rate is crucial to avoid pulling excessive telemetry data, which could result in buffer overflow. In the egress pipeline, InvisiFlow randomly drops generated seed packets, with the probability of dropping packets equal to the buffer usage. Moreover, if a switch's buffer usage exceeds a certain threshold (*e.g.*, > 95%), it will revert to blindly pushing out telemetry packets to neighboring switches, as in prior work [54]. While push-based sending carries risks for telemetry packets, extreme buffer usage suggests they would likely have been dropped otherwise. We find that this approach effectively prevents the loss of telemetry data, particularly in scenarios with asymmetric bandwidth usage.

4.2 Server Operation

Two types of servers participate in InvisiFlow: those that collect telemetry data and those that serve as temporary storage.

Collectors. To get telemetry data from switches, collectors simply send pull requests to the network in the same manner as switches. We set the buffer size recorded in the pull request to 0. Based on §4.1, neighbor switches will send any available telemetry packets in response to these pull requests. Note that it also works if there are multiple collectors, and the telemetry data will flow to any one of the collectors.

Note that collectors also have the option to *not* send pull requests if, for instance, they lack sufficient CPU resources or storage bandwidth to keep up with the current inflow. Alternatively, collectors can offload the collection tasks to the last-hop switch as in prior works [43], *i.e.*, the last-hop switch is responsible for generating pull requests and directly writes the telemetry data to servers' memory throughput Remote Direct Memory Access (RDMA).

Available servers. To further minimize telemetry data loss, InvisiFlow can optionally explore available memory/storage in servers to store telemetry packets. We assume that the

Resource	Baseline	Marginal Increase
Gateway	11.46%	2.08%
VLIW Actions	4.69%	0.78%
Hash Bit	5.07%	0.46%
Exact Match Input xbar	3.58%	0.39%
Meter ALU	14.58%	0.00%
SRAM	1.77%	0.00%
Stages	7	0

Table 2: Additional H/W resources used by InvisiFlow. Baseline is the low-priority UDP using OrbWeaver [21] to generate telemetry packets.

memory/storage capacity in these available servers is much larger than that of switches. Similar to collectors, available servers join InvisiFlow by sending pull requests. The difference is that we set the buffer usage in the pull request to a constant value α (*e.g.*, 50%). These available servers can pull telemetry packets from their neighboring switches if the buffer usage of the switch is larger than α. Otherwise, the neighbor switches pull telemetry packets from these available servers. The setting of α determines the usage rate of the memory/storage in available servers and the possibility of data loss as shown in §6.2. A larger α may lead to a larger possibility of data loss but a lower usage rate of the memory/storage in available servers, and vice versa. The available servers can also dynamically adjust α based on their available memory and leave enough space for user applications.

5 Implementation

We implement an InvisiFlow prototype on a testbed with two Wedge100BF-32X switches. Servers are equipped with a Mellanox ConnectX-5 100 Gbps NIC, dual-socket AMD EPYC 7313 16-Core Processor, and 64 GB memory.

InvisiFlow servers. Per §4.2, each InvisiFlow-server (*i.e.*, a collector or a free server) sends out pull requests if there is available space and processes pull requests or telemetry packets from the network. In our testbed, we implement the collectors and free servers using DPDK (v21.11).

InvisiFlow switches. We implement InvisiFlow's switch component in P4-16. Our implementation leverages the built-in packet generation and strict prioritization features present in today's switches. In total, the data plane implementation consists of around 600 lines of P4 code. Our implementation scales to today's multi-pipeline switching architectures.

Egress pipeline register buffer for telemetry packets: Similar to PayloadPark [55], we use an array of registers in the egress pipeline to store telemetry packets. Due to the constraints on the number of stages and the number of bytes read/written per stage, the size of telemetry packets should be small (*e.g.*, <160 B) as in [55]. This amount is sufficient for most applications [5, 27, 28] and even allows for multiple telemetry records within a single packet in many cases. Larger telemetry messages can be handled with per-hop segmentation of telemetry packets. To improve the goodput of telemetry data, we omit the IP and UDP header and include only Ethernet. Different *EtherType*s distinguish different types of packets, *e.g.*, user packets, pull requests, or telemetry packets.

Switch-local teleport of telemetry data: Besides front-panel ports, modern switches also have per-pipeline internal ports for pktgen traffic and recirculation. InvisiFlow consumes the bandwidth of one of these ports to move the telemetry data to the egress pipeline, as discussed in §4.1. Note that InvisiFlow does not recirculate any telemetry packets nor suffer from recirculation overhead. The egress pipeline drops telemetry packets from the ingress pipeline immediately after storing the telemetry data in the egress pipeline register.

Hardware resource usage: Table 2 summarizes the key resources required by our prototype. Compared with low-priority UDP, which uses OrbWeaver [21] to generate low-priority telemetry packets, the additional resource used by InvisiFlow is small: 2.08% additional gateways and 0.78% additional VLIW actions to process the telemetry packets. The ALU and SRAM are mainly used to store and read/write telemetry data in the egress pipeline.

6 Evaluation

We evaluate InvisiFlow using a combination of testbed experiments and large-scale ns-3 simulations [56]. Our evaluation focuses on the following three key questions.

- Can InvisiFlow minimize the accuracy degradation on the telemetry application (R1)?
- Can InvisiFlow simultaneously minimize the overhead of transmitting telemetry data (R2)?
- Is InvisiFlow robust to different settings and workloads?

6.1 Simulation Methodology

Topology and workload. We simulate a FatTree [45] topology with 9 core switches and 4 pods that, combined, have 12 aggregation switches, 12 top-of-rack (ToR) switches and 144 servers. All links have a capacity of 100 Gbps, leading to a 4:1 oversubscription ratio similar to prior work [57–59]. The propagation delay for all links is 1 µs, resulting in a maximum base RTT of 12 µs. Switch buffer sizes are set proportionally to the bandwidth-buffer ratio of Intel Tofino switches [60], following prior research [58, 59]. Specifically, each switch has approximately 2 MB of buffer for up to 600 Gbps of user traffic and 170 KB for up to 50 Gbps of telemetry traffic[2]. These buffers are shared by all ports on the same switch and are fully isolated to prevent any impact on user traffic. For DCTCP, we set $K = 65$ based on [59, 61]. User traffic is load-balanced using ECMP over the 5-tuple.

Experiments are performed with one of two workloads: a Hadoop workload (FB_Hadoop) and a machine-learning (ML) workload. Details of the workloads can be found in Appendix B. Unless otherwise specwified, FB_Hadoop is the default setting.

[2] In experiments, the total telemetry data generated is generally <30 Gbps.

Stressed scenarios. In addition to the default setting, we introduce two other scenarios to test InvisiFlow's robustness.

- *Load imbalance:* We simulate a load imbalance scenario caused by a faulty configuration, where ECMP is configured to hash based only on the destination IP, which causes packets to a single destination to use only one path, leading to high load imbalance, as discussed in prior work [62].
- *Asymmetric topology:* We also stress the collector under an asymmetric topology with failed fabric links [20, 63]. We reduce the capacity of two links to 50 Gbps. These links include a Core-Aggregation link and an aggregation-ToR link, both toward the collector.

Telemetry sources, channels, and sinks. Next, we describe the evaluated configurations of the three components of the telemetry frameworks depicted in §2.1.

Sources: Four telemetry applications run concurrently between 20% and 80% of the simulation time.

- *Path tracing (NetSeer [5]):* Following [5], each switch maintains a record of user flows passing through it. When encountering a new flow, the switch stores the tuple (switch ID, flow ID, TTL value) in its buffer. The collector uses the TTL value to calculate the hop ID of each node and reconstruct the entire path of each flow.
- *Load imbalance detection (NetFlow [36]):* Following [36], every switch records the number of bytes passing through each egress port in the data plane. The switch then stores the tuple (switch ID, port ID, time, # of bytes) in the buffer and reports the results (by default) every 5 µs. The collector aggregates the byte counts to calculate each switch's load. It measures the load imbalance rate as the maximum load relative to the minimum load among switches.
- *Per-flow accounting (ApproSync [31]):* We implement ApproSync [31] with Count-Min Sketch (CMS) [32] to track the size per-flow. In the CMS, we use 3 arrays of 2^{17} counters per array. ApproSync sends sketches to the collector by checking the divergence of each counter in the sketch. Divergence is quantified using relative error, with a threshold based on an expected bandwidth of 10 Gbps.
- *Packet drop notification (NetSeer [5]):* Similar to [5], when a user packet drop occurs, the switch stores the tuple (switch ID, flow ID) in the buffer.

Channels: We compare InvisiFlow (IF) with three communication substrates for telemetry data:

- *Default-priority UDP:* Prior telemetry systems like NetFlow [36], sFlow [42], and NetSight [29] often use default-priority UDP to send telemetry data. We implement default-priority UDP in postcard mode [29]: When there is telemetry data to be sent, the switch generates a telemetry packet using mirroring and sends it to the collector. Telemetry packets are assigned the same priority as user traffic. If multiple shortest paths are available to the collector, we use packet spraying [51] for load balancing.
- *Low-priority UDP:* Some telemetry systems like Everflow [28] and Planck [27] assign low-priority to telemetry packets to avoid impacting user traffic. To implement low-priority UDP, we use OrbWeaver [21] to generate low-priority telemetry packets. These packets are forwarded to the next hop when there is available bandwidth. As before, we use packet spraying if there are multiple shortest paths.
- *Pull-based transmission:* As an ablation study, we include pull-based transmission without using congestion gradients. Each switch generates pull requests only to switches farther from the collector, so that telemetry packets follow the shortest paths. A switch pulls only when its telemetry buffer usage is below 50%. We reserve some buffer capacity to prevent overflow, as there could be locally generated or late telemetry data due to low-priority transmission.

Sinks: In our experiments, one of the 144 servers is always reserved as a collector server. When additional sinks are required, they are taken from other clusters. We assume that every switch is pre-configured with their addresses.

6.2 Telemetry Application Performance

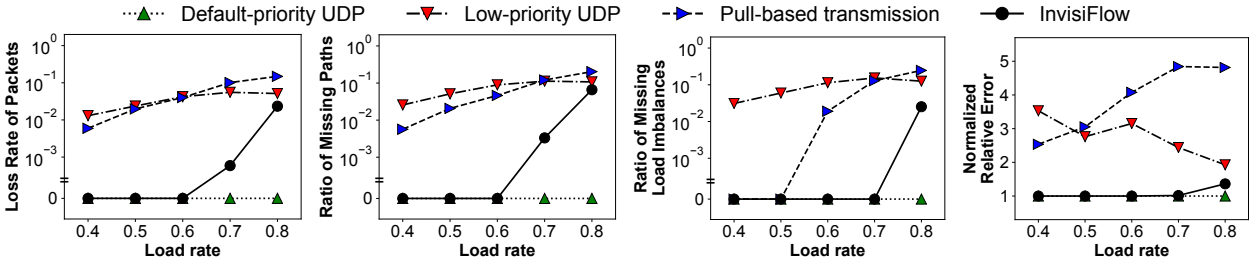
This section compares the impact of InvisiFlow to alternative communication substrates on telemetry applications, highlighting InvisiFlow's ability to minimize accuracy degradation, particularly for low-priority transmissions.

Performance metrics. We consider the following metrics:

- *Ratio of missing paths:* Following [3], the collector reports all paths it receives. If the collector fails to capture one or more switch IDs along a path, we mark the path missed.
- *Ratio of missing load imbalance events:* We calculate the ratio of missing events in core switches for load imbalance detection as in [10]. A missed event occurs when the collector cannot identify a period where the imbalance rate at the core switches exceeds 2.
- *Normalized relative error of the flow size:* As in [31], all core switches estimate the number of packets per flow using sketches before sending them to the collector. To quantify the impact of telemetry loss, we calculate the average relative error of those estimates normalized to the case without loss (i.e., we exclude errors introduced by ApproSync and the Count-Min sketch). The errors for heavy hitters (*i.e.*, $> 10^3$ packets) are shown in §C.

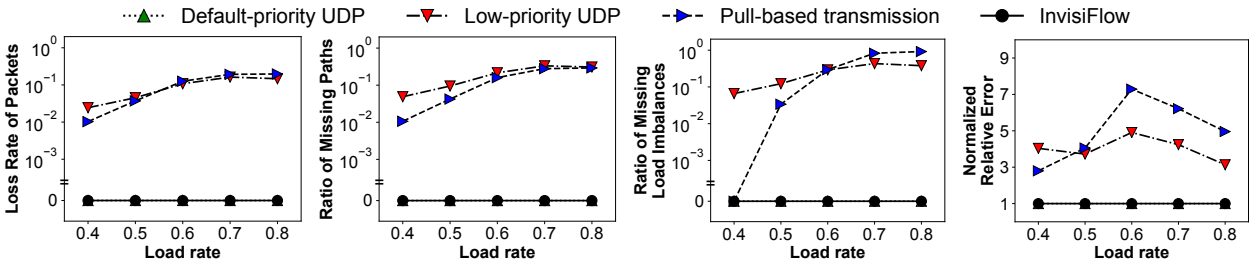
Since user packet drops are rare in our setup, we do not present the loss rate for packet drop notifications. Ground truth results are captured with the help of extensive out-of-band logging in the simulator.

Default setting (Figure 6). Figure 6 illustrates the performance of different telemetry applications under varying load rates. InvisiFlow maintains a zero loss rate for telemetry packets when the load rate is below 70%. Consequently, the ratio of missing paths and load imbalances are also 0, and the normalized relative error is 1 below this threshold. However, InvisiFlow has to drop telemetry packets when the load ex-



(a) Telemetry packet loss (b) Path loss for NetSeer (c) Imbalance loss for NetFlow (d) Flow error for ApproSync

Figure 6: Default setting. InvisiFlow maintains a zero loss rate when the load rate is below 70%.



(a) Telemetry packet loss (b) Path loss for NetSeer (c) Imbalance loss for NetFlow (d) Flow error for ApproSync

Figure 7: Load imbalance. InvisiFlow maintains a zero telemetry packet drop rate even under an 80% load, whereas the loss rates of low-priority UDP and pull-based transmission increase compared to the default setting.

ceeds 70% due to limited available bandwidth for low-priority transmission, which could be smaller than the generation rate of the telemetry packets. The performance of pull-based transmissions deteriorates under heavy loads compared to low-priority UDP because it cannot fully utilize the buffer and bandwidth when the available bandwidth is insufficient for the telemetry data throughput. Default-priority UDP does not drop packets even under 80% load, but increases the FCT by approximately 26.4%, as discussed in §6.3.

At a 70% load, InvisiFlow's ratio of missing paths is approximately 33.8× and 36.3× lower than that of low-priority UDP and pull-based baselines, respectively. The gap between them narrows under an 80% load, as there is limited room to reduce loss due to insufficient available bandwidth for low-priority packets at such a high load rate. However, InvisiFlow still shows a 1.6× and 3.1× improvement over low-priority UDP and pull-based baselines, respectively. Regarding the relative error of all flows, InvisiFlow at a 70% load exhibits a reduction of approximately 2.4× and 4.8× compared to low-priority UDP and pull-based baselines, respectively. Normalized relative errors may decrease with higher load rates due to increased error in the original Count-Min sketch caused by higher traffic volumes.

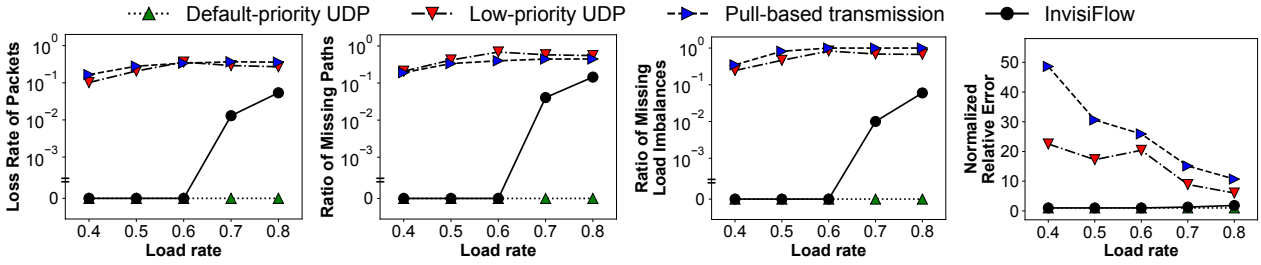
Load imbalance (Figure 7). In this scenario, InvisiFlow maintains a zero telemetry loss rate even under an 80% load, whereas the loss rates of low-priority UDP and pull-based transmission increase compared to the default setting. This is because the load imbalance reduces the throughput of user traffic, potentially leaving more available bandwidth for telemetry data. However, low-priority UDP and pull-based baselines fail to fully utilize this additional bandwidth as they remain constrained to the shortest path.

At a 70% load, the ratio of missing paths for InvisiFlow is 0, while it is 33.2% and 27.9% for low-priority UDP and pull-based baselines, respectively. Furthermore, the normalized relative error of all flows for InvisiFlow is 1, while it is 4.2 and 6.2 for low-priority UDP and pull-based, respectively.

Asymmetric topology (Figure 8). In an asymmetric topology, the loss rates of low-priority UDP and pull-based transmission experience significant increases. In contrast, InvisiFlow only drops telemetry packets when the load rate exceeds 70%. Specifically, the ratio of missing paths for low-priority UDP often exceeds 40%, and for the pull-based baseline, it often exceeds 30%. Furthermore, the normalized relative error of all flows for low-priority UDP and pull-based baselines can reach up to 22.5 and 48.6, respectively.

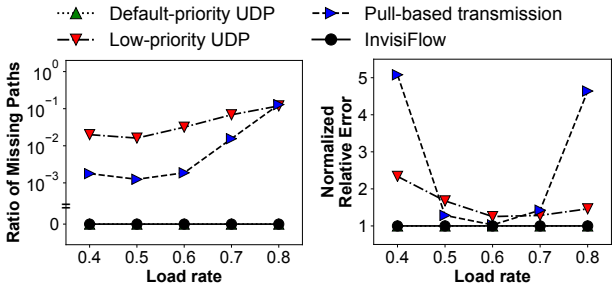
ML workloads (Figure 9). In ML workloads, InvisiFlow has a zero telemetry packet drop rate even under an 80% load, while low-priority UDP and pull-based transmission lose approximately 11.9% and 12.9% of paths at an 80% load, respectively. Furthermore, at a 70% load, the relative error of all flows for InvisiFlow is approximately 1.5× and 4.6× lower than that of low-priority UDP and pull-based, respectively.

Different buffer sizes (Figure 10). In addition to the default buffer size of 170 KB, we experiment with different buffer sizes under the default setting. We observe that InvisiFlow achieves a 0% loss rate when the buffer size exceeds 400 KB, because the congestion gradient-based transmission efficiently utilizes all available telemetry buffers across the network. In contrast, low-priority UDP and pull-based transmission lose around 5.8% and 9.8% of paths, respectively, even with a 500 KB buffer. With a 500 KB buffer,



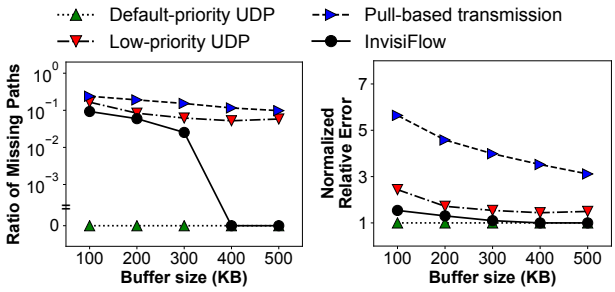
(a) Telemetry packet loss (b) Path loss for NetSeer (c) Imbalance loss for NetFlow (d) Flow error for ApproSync

Figure 8: Asymmetric topology. InvisiFlow only drops telemetry packets when the load rate exceeds 0.7. The ratio of missing paths for low-priority UDP often exceeds 40%, and for the pull-based baseline, it often exceeds 30%.



(a) Path loss for NetSeer (b) Flow error for ApproSync

Figure 9: ML workload (Default setting).



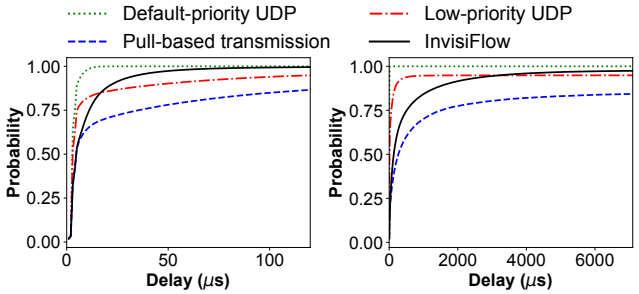
(a) Path loss for NetSeer (b) Flow error for ApproSync

Figure 10: Different buffer sizes (80% load).

the relative error of all flows for InvisiFlow is approximately 1.5× and 3.1× lower than that of low-priority UDP and pull-based baselines, respectively.

CDF for delay of telemetry packets (Figure 11). Figure 11 illustrates the end-to-end delay of the telemetry packets. We calculate the delay by subtracting the timestamp at which the collector receives the packet from the timestamp at which the telemetry packet is generated. In case of packet loss, we consider the delay infinite. Notably, InvisiFlow significantly improves tail latency compared to other low-priority transmissions. Under a 40% load, the 99% latency of InvisiFlow is approximately 80 μs, which is around 3.4× and 10.9× lower than that of low-priority UDP and pull-based baselines, respectively. Under an 80% load, InvisiFlow increases the delay to avoid both telemetry packet loss and impact on user traffic, with a 99% latency of around 4.9 ms.

Using multiple servers (Table 3). Using multiple collectors or additional available servers as temporary storage can further reduce telemetry data loss. When using two collectors,



(a) 40% load rate. (b) 80% load rate.

Figure 11: Delay of telemetry packets (Default setting).

telemetry data can be directed to any available collector, enabling a 0% loss rate. When using one available server, the threshold of the buffer usage determines the storage usage in available servers and the data loss rate. A higher threshold increases the risk of data loss but reduces the storage usage in available servers, and vice versa. Setting the threshold to 25% allows zero loss rate while utilizing up to 13.63 MB of storage for telemetry packets under the default setting. Unlike using multiple collectors where telemetry data can be distributed, using available servers results in all telemetry data being sent to the single designated server without further aggregation.

6.3 Impact on User Traffic

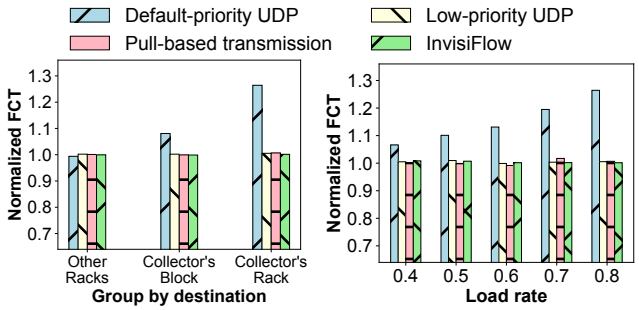
In this section, we compare InvisiFlow with other communication substrates and show that InvisiFlow achieves near-zero overhead to user traffic. The experimental setup is consistent with that in §6.2.

Metric. We evaluate the flow completion time (FCT) of user traffic initiated between 20% and 80% of the simulation time (*i.e.*, when the telemetry applications are running). The FCT is normalized to the case where no telemetry data is generated.

Default setting (Figure 12). In Figure 12a, we group the FCT by the destination of the user traffic and calculate the ratio. We find that user traffic destined for the rack of the collector is primarily influenced by the telemetry data under the default setting. Assigning telemetry data default priority results in an FCT increase of ∼26.4% under an 80% load. In contrast, InvisiFlow and low-priority baselines exhibit minimal overhead, often <1%. Figure 12b shows the ratio under different load rates, showing that the overhead of InvisiFlow and

Default Setting	Telem. Loss	Peak Avail. Util
1 Collector	2.34%	N/A
+ 1 Collector	0.00%	N/A
+ 1 Available Server *Threshold: 50%	1.94%	2.27 MB
+ 1 Available Server *Threshold: 25%	0.00%	13.63 MB
Asymmetric topology	**Telem. Loss**	**Peak Avail. Util**
1 Collector	5.38%	N/A
+ 1 Collector	0.00%	N/A
+ 1 Available Server *Threshold: 50%	1.79%	10.92 MB
+ 1 Available Server *Threshold: 25%	0.00%	25.83 MB

Table 3: Telemetry packet loss rate and peak storage utilization of available servers when using multiple servers for telemetry data collection. We evaluate the default and asymmetric topologies under 80% load.



(a) 80% load rate. **(b) In Collector's rack.**

Figure 12: Normalized FCT (Default setting).

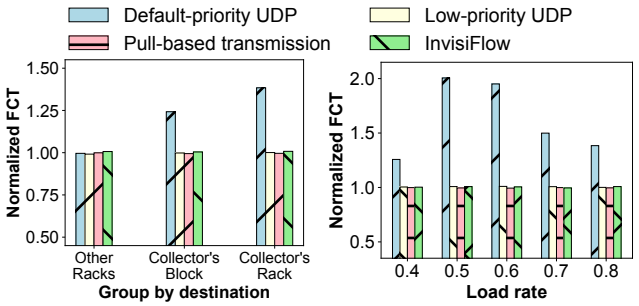
low-priority remains minimal, while the overhead of default-priority UDP increases with higher load rates.

Asymmetric topology (Figure 13). As depicted in Figure 13a, user traffic not only to the rack but also to the block containing the collector is significantly influenced by telemetry data under the asymmetric topology. For default-priority UDP, the FCT of user traffic to the rack increases by ∼38.4%, and the block by ∼25.2%. In contrast, InvisiFlow and low-priority baselines maintain near-zero overhead (<1%). Figure 13b illustrates that under a 50% load, the FCT of user traffic is doubled. The normalized FCT may decrease with increasing load rates due to user traffic causing more congestion.

6.4 Testbed Experiments

Topology (Figure 14a). We emulate a 2-tier leaf-spine topology in our testbed. With two physical switches, similar to [11], we divide one 128-port physical switch into four virtual leaf switches and the other into two virtual spine switches. The virtual switches are connected with 10 Gbps links. In the data plane, every logical switch has its isolated buffer of telemetry packets and runs InvisiFlow independently.

Setup. We emulate the case that a link to the collector is fully



(a) 80% load rate. **(b) In Collector's rack.**

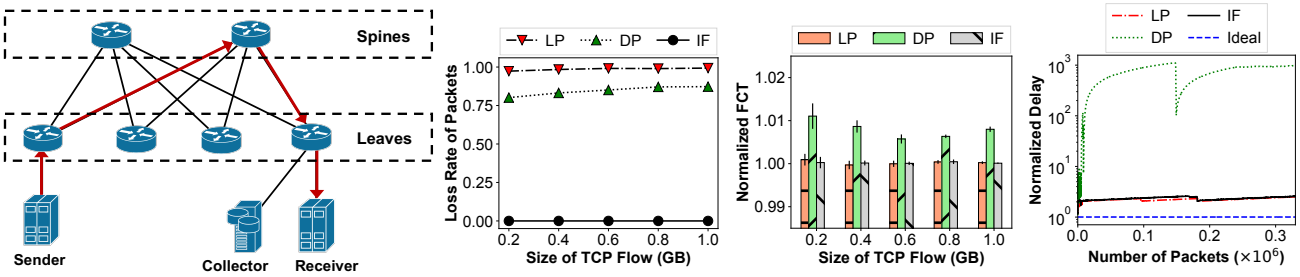
Figure 13: Normalized FCT (Asymmetric topology).

occupied by user traffic. Specifically, as shown in Figure 14a, we build a TCP connection between two servers and send traffic. Then, we measure the statistics in the right spine switch, which is the main bottleneck for telemetry data transmission. We use the load imbalance detection as the application and compare InvisiFlow (IF) with the low-priority UDP (LP) and the default-priority UDP (DP) as above.

Loss rate of telemetry packets (Figure 14b). In Figure 14b, we measure the loss rate of telemetry data in the right spine switch. We can find that the loss rate for low-priority UDP is higher than 97%. Because the only shortest path to the collector is fully occupied by the user traffic, low-priority UDP can hardly send telemetry data. In contrast, the loss rate of InvisiFlow is always 0 because InvisiFlow can find other available paths to send telemetry data. Note that the loss rate of default-priority UDP is also higher than 80% and increases over the size of TCP flows. It may be due to the switch architecture, *i.e.*, mirroring could fail when the corresponding egress port is fully utilized by user traffic.

Impact on user traffic (Figures 14c and 14d). In Figure 14c, we measure the ratio between the flow completion time (FCT) of user traffic with telemetry data against that without any telemetry. The overhead incurred by low-priority UDP and InvisiFlow is negligible (<0.1%), while the overhead of the default-priority UDP is higher (≈0.8%). The overhead of the default-priority UDP is mitigated because the switch already drops over 80% of telemetry packets when user traffic fully utilizes the link, as illustrated in Figure 14b.

In Figure 14d, we measure the delay of user packets within the right spine switch over time. Specifically, we calculate the switch delay by subtracting the timestamp at the egress pipeline from the timestamp at the ingress pipeline. The ideal delay in the figure represents the delay without any telemetry data. We can find that the default-priority UDP will gradually increase the delay of user packets over time, leading to the queue becoming full and subsequent packet drops. The delay for default-priority UDP could be up to $10^3\times$ larger than the ideal delay. Conversely, the delay of low-priority UDP and InvisiFlow is similar, which keeps around $2\times$ as the ideal delay. In other words, the delay of default-priority UDP could be up to $500\times$ larger than that of InvisiFlow. The reason for the additional delay ($1\times$ more normalized delay) in low-



(a) Testbed topology. (b) Telemetry packet loss. (c) Flow completion time. (d) Queuing delay.

Figure 14: Testbed experiments.

priority UDP and InvisiFlow might be that the switch will start strict prioritization only after there is a user packet waiting in the queue. However, it will not influence the overall flow completion time as shown in Figure 14c.

7 Discussion

Limitations of InvisiFlow. We note that InvisiFlow, in its effort to minimize the impact on user traffic, may result in drops and arbitrary delays of telemetry packets, particularly if all ports of the switch are fully utilized by user traffic. Under an extreme case where none of the switches have available bandwidth, all low-priority transmissions, including InvisiFlow, would have to drop packets. However, such scenarios are rare in modern high-speed networks. As demonstrated in §6.2, even under 80% load, InvisiFlow can reduce missing paths by around 1.63× and 3.08× compared to low-priority UDP and pull-based baselines, respectively. If a portion of telemetry traffic is essential, it can always fall back on standard transport mechanisms. InvisiFlow may also struggle with networks that need a large number of distinct collector types, as it requires the use of a separate buffer allocation for every type. Switching ASICs with more flexible memory access patterns may alleviate this limitation [64, 65].

Low-latency transmission. In some cases, telemetry reports may only be valuable if collected in a timely manner. Heretofore, InvisiFlow was presented as a framework that minimizes loss but does not bound the latency of reports. To support time-sensitive reports, we can adapt InvisiFlow by restricting sending to only hops that are strictly closer to the eventual destination. Packets would have fewer options for forwarding, but prior work on progressive routing has shown that adaptive routing policies like InvisiFlow's are still very effective [66].

Generality to other applications. InvisiFlow's abstraction to exploit network-wide under-utilized bandwidth is also applicable to other application amenable to low-priority transmission. As discussed above, InvisiFlow cannot (1) guarantee the delivery and (2) support too many distinct destinations. To address these problems, we can (1) ask the above application to handle delivery or add a layer above InvisiFlow to guarantee delivery as TCP and (2) group multiple destinations into one for simplicity, similar to how we group multiple IP addresses into an IP prefix. We leave the extension to user applications as future work.

8 Related Work

Applications of congestion gradients. Congestion gradients (*i.e.*, max-weight scheduling) have mainly been studied in a theoretical context [49, 50]. The mathematical optimality properties of congestion gradients have motivated experimental demonstrations of its use on wireless sensor networks [67, 68]. Some work [69] also applies it to urban traffic networks. However, these works focus on entirely different scenarios, so they need to consider many different factors (*e.g.*, lossy links in wireless networks) and implement the systems on different platforms (*e.g.*, CPU in sensors).

Per-packet multi-path routing. A commonly studied per-packet multi-path routing technique is packet spraying [51]. However, as shown in §6.2, the loss rate of low-priority UDP remains high even when utilizing packet spraying. This is because packet spraying assumes symmetry topology, while the available bandwidth for low-priority transmission is often asymmetric and varies largely over time. Furthermore, there are also some other per-packet multi-path routing protocols proposed [70, 71], primarily for wireless networks. As discussed above, they focus on entirely different scenarios and are mainly designed to achieve different objectives (*e.g.*, energy efficiency, security, and reliability).

Explore external memory in servers. Prior work has suggested system architectures that allow switches to utilize external memory on servers [52, 53]. The switches can directly access DRAM on servers via RDMA. However, similar to default-priority UDP, such a way is not friendly to achieve near-zero overhead on user traffic. In addition, [52, 53] need to pre-define the memory allocated in servers, while InvisiFlow can dynamically adjust the size of the memory needed.

9 Conclusion

Existing transmission methods for telemetry data often have to choose between overhead to user traffic and sustainable throughput of telemetry. To achieve both, we introduce InvisiFlow, a low-priority communication substrate utilizing opportunistic sending and congestion gradients. Our experimental results show that InvisiFlow can support high sustainable throughput with minimal impact on user traffic across multiple different applications and settings.

Acknowledgments

We thank the anonymous reviewers for their thorough comments and feedback. This work was funded in part by NSF grant CNS-1845749. The work was also partially supported by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART").

References

[1] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, David A. Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *SIGCOMM 2015*, pages 139–152. ACM, 2015.

[2] Tal Mizrahi, Gidi Navon, Giuseppe Fioccola, Mauro Cociglio, Mach (Guoyi) Chen, and Greg Mirsky. AM-PM: efficient network telemetry using alternate marking. *IEEE Netw.*, 33(4):155–161, 2019.

[3] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with pathdump. In *OSDI 2016*, pages 233–248. USENIX Association, 2016.

[4] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Cherrypick: tracing packet trajectory in software-defined datacenter networks. In *SOSR '15*. ACM, 2015.

[5] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. Flow event telemetry on programmable data plane. In *SIGCOMM '20*, pages 76–89. ACM, 2020.

[6] Naga Praveen Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: scalable load balancing using programmable data planes. In *SOSR 2016*, page 10. ACM, 2016.

[7] Naveen Kr Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *NSDI '18)*, pages 1–16, 2018.

[8] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive programmable switches. In *SIGCOMM '20*, page 296–309. ACM, 2020.

[9] Yinda Zhang, Peiqing Chen, and Zaoxing Liu. Octosketch: Enabling real-time, continuous network monitoring over multiple cores. In *NSDI 2024*. USENIX Association, 2024.

[10] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *IMC '17*, page 78–85. Association for Computing Machinery, 2017.

[11] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *SIGCOMM 2018*, pages 402–416. ACM, 2018.

[12] Soudeh Ghorbani, Zibin Yang, P Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. Drill: Micro load balancing for low-latency data center networks. In *SIGCOMM*, pages 225–238, 2017.

[13] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. Cocosketch: high-performance sketch-based measurement over arbitrary partial key query. In *ACM SIGCOMM 2021 Conference*, pages 207–222. ACM, 2021.

[14] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: adaptive and fast network-wide measurements. In *SIGCOMM 2018*, pages 561–575. ACM, 2018.

[15] Anup Agarwal, Zaoxing Liu, and Srinivasan Seshan. Heterosketch: Coordinating network-wide monitoring in heterogeneous and dynamic networks. In *NSDI 2022*, pages 719–741. USENIX Association, 2022.

[16] Haifeng Sun, Qun Huang, Jinbo Sun, Wei Wang, Jiaheng Li, Fuliang Li, Yungang Bao, Xin Yao, and Gong Zhang. Autosketch: Automatic sketch-oriented compiler for query-driven network telemetry. In *NSDI 2024*. USENIX Association, 2024.

[17] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: high precision congestion control. In *SIGCOMM 2019*, pages 44–58. ACM, 2019.

[18] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, Lawrence J Wobker, et al. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, volume 15, pages 1–2, 2015.

[19] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minlan Yu, and Michael Mitzenmacher. PINT: probabilistic in-band network telemetry. In *SIGCOMM '20*, pages 662–680. ACM, 2020.

[20] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas E. Anderson. F10: A fault-tolerant engineered network. In *NSDI 2013*, pages 399–412. USENIX Association, 2013.

[21] Liangcheng Yu, John Sonchack, and Vincent Liu. Orbweaver: Using IDLE cycles in programmable networks for opportunistic coordination. In *NSDI 2022*, pages 1195–1212. USENIX Association, 2022.

[22] Yiran Lei, Liangcheng Yu, Vincent Liu, and Mingwei Xu. Printqueue: performance diagnosis via queue measurement in the data plane. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 516–529, 2022.

[23] Snmp. https://datatracker.ietf.org/doc/html/rfc1157.

[24] Framework for in-situ flow information telemetry. https://datatracker.ietf.org/doc/draft-song-opsawg-ifit-framework/.

[25] Inband flow analyzer. https://datatracker.ietf.org/doc/html/draft-kumar-ippm-ifa.

[26] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, and Nicholas Zhang. Lightguardian: A full-visibility, lightweight, in-band telemetry system using sketchlets. In *NSDI 2021*, pages 991–1010. USENIX Association, 2021.

[27] Jeff Rasley, Brent E. Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John B. Carter, and Rodrigo Fonseca. Planck: millisecond-scale monitoring and control for commodity networks. In *SIGCOMM'14*, pages 407–418. ACM, 2014.

[28] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert G. Greenberg, Guohan Lu, Ratul Mahajan, David A. Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. In *SIGCOMM 2015,*, pages 479–491. ACM, 2015.

[29] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI 2014*, pages 71–85. USENIX Association, 2014.

[30] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. dshark: A general, easy to program and scalable framework for analyzing in-network packet traces. In *NSDI 2019*, pages 207–220. USENIX Association, 2019.

[31] Xiang Chen, Qun Huang, Dong Zhang, Haifeng Zhou, and Chunming Wu. Approsync: Approximate state synchronization for programmable networks. In *ICNP 2020*, pages 1–12. IEEE, 2020.

[32] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 2005.

[33] Xiaoqi Chen, Shir Landau Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *SIGCOMM '20*, pages 226–239. ACM, 2020.

[34] Nguyen Van Tu, Jonghwan Hyun, Ga Yeon Kim, Jae-Hyoung Yoo, and James Won-Ki Hong. Intcollector: A high-performance collector for in-band network telemetry. In *CNSM 2018*, pages 10–18. IEEE Computer Society, 2018.

[35] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *NSDI 2019*, pages 421–436. USENIX Association, 2019.

[36] Benoit Claise. Cisco systems netflow services export version 9. *RFC*, 3954:1–33, 2004.

[37] Generating netflow records. https://www.cisco.com/c/en/us/td/docs/dcn/nexus-data-broker/392/use-case/generate-netflow-records.html.

[38] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *SIGCOMM 2016*, pages 101–114. ACM, 2016.

[39] Qun Huang, Patrick P. C. Lee, and Yungang Bao. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In *SIGCOMM 2018*, pages 576–590. ACM, 2018.

[40] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM 2015*, pages 123–137. ACM, 2015.

[41] Siyuan Sheng, Qun Huang, and Patrick P. C. Lee. Deltaint: Toward general in-band network telemetry with extremely low bandwidth overhead. In *ICNP 2021*, pages 1–11. IEEE, 2021.

[42] Peter Phaal, Sonia Panchen, and Neil McKee. Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks. *RFC*, 3176:1–31, 2001.

[43] Jonatan Langlet, Ran Ben Basat, Gabriele Oliaro, Michael Mitzenmacher, Minlan Yu, and Gianni Antichi. Direct telemetry access. In *SIGCOMM 2023*, pages 832–849, 2023.

[44] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, nov 1984.

[45] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM 2008*, pages 63–74. ACM, 2008.

[46] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: experience with a globally-deployed software defined wan. In *SIGCOMM 2013*, pages 3–14. ACM, 2013.

[47] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM 2013*, pages 15–26. ACM, 2013.

[48] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC 2010*, pages 267–280. ACM, 2010.

[49] Leandros Tassiulas and Anthony Ephremides. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. In *29th IEEE Conference on Decision and Control*, pages 2130–2132. IEEE, 1990.

[50] Baruch Awerbuch and Frank Thomson Leighton. A simple local-control approximation algorithm for multicommodity flow. In *34th Annual Symposium on Foundations of Computer Science, Palo Alto, California, USA, 3-5 November 1993*, pages 459–468. IEEE Computer Society, 1993.

[51] Advait Abhay Dixit, Pawan Prakash, Y. Charlie Hu, and Ramana Rao Kompella. On the impact of packet spraying in data center networks. In *INFOCOM 2013*, pages 2130–2138. IEEE, 2013.

[52] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. TEA: enabling state-intensive network functions on programmable switches. In *SIGCOMM '20*, pages 90–106. ACM, 2020.

[53] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostic, and Marco Chiesa. A high-speed stateful packet processing approach for tbps programmable switches. In *NSDI 2023*, pages 1237–1255. USENIX Association, 2023.

[54] Kyriakos Zarifis, Rui Miao, Matt Calder, Ethan Katz-Bassett, Minlan Yu, and Jitendra Padhye. DIBS: just-in-time congestion mitigation for data centers. In *EuroSys 2014*, pages 6:1–6:14. ACM, 2014.

[55] Swati Goswami, Nodir Kodirov, Craig Mustard, Ivan Beschastnikh, and Margo I. Seltzer. Parking packet payload with P4. In *CoNEXT '20*, pages 274–281. ACM, 2020.

[56] Network simulator 3. https://www.nsnam.org/.

[57] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa H. Ammar, Ellen W. Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani, and Amin Vahdat. Annulus: A dual congestion control loop for datacenter and WAN traffic aggregates. In *SIGCOMM '20*, pages 735–749. ACM, 2020.

[58] Vamsi Addanki, Oliver Michel, and Stefan Schmid. Powertcp: Pushing the performance limits of datacenter networks. In *NSDI 2022*, pages 51–70. USENIX Association, 2022.

[59] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. ABM: active buffer management in datacenters. In *SIGCOMM '22*, pages 36–52. ACM, 2022.

[60] Barefoot tofino: World's fastest p4-programmable ethernet switch asics. https://barefootnetworks.com/products/brief-tofino/.

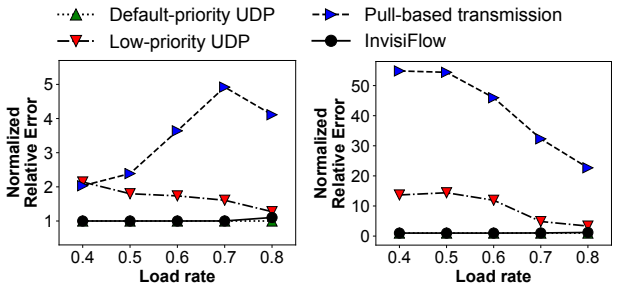
[61] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM 2010*, pages 63–74. ACM, 2010.

[62] Jie Li, Shi Zhou, and Vasileios Giotsas. Performance analysis of multipath BGP. In *INFOCOM Workshops 2021*, pages 1–6. IEEE, 2021.

[63] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: distributed congestion-aware load balancing for datacenters. In *SIGCOMM'14*, pages 503–514. ACM, 2014.

[64] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. drmt: Disaggregated programmable switching. In *SIGCOMM 2017*, pages 1–14. ACM, 2017.

[65] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. Using trio: juniper networks' programmable chipset - for emerging in-network applications. In *SIGCOMM 2022*, pages 633–648. ACM, 2022.



(a) Default setting **(b) Asymmetric topology**

Figure 15: Flow error for heavy hitters.

[66] Jose Duato, Sudhakar Yalamanchili, and Ni Lionel. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers Inc., 2002.

[67] Avinash Sridharan, Scott Moeller, and Bhaskar Krishnamachari. Making distributed rate control using lyapunov drifts a reality in wireless sensor networks. In *WIOPT 2008*, pages 452–461. IEEE, 2008.

[68] Scott Moeller, Avinash Sridharan, Bhaskar Krishnamachari, and Omprakash Gnawali. Routing without routes: the backpressure collection protocol. In *IPSN 2010*, pages 279–290. ACM, 2010.

[69] Xingmin Wang, Yafeng Yin, Yiheng Feng, and Henry X Liu. Learning the max pressure control for urban traffic networks considering the phase switching loss. *Transportation Research Part C: Emerging Technologies*, 140:103670, 2022.

[70] Jenn-Yue Teo, Yajun Ha, and Chen-Khong Tham. Interference-minimized multipath routing with congestion control in wireless sensor network for high-rate streaming. *IEEE Trans. Mob. Comput.*, 7(9):1124–1137, 2008.

[71] Moufida Maimour. Maximally radio-disjoint multipath routing for wireless multimedia sensor networks. In *WMuNeP'07, 2008*, pages 26–31. ACM, 2008.

[72] Sudarsanan Rajasekaran, Manya Ghobadi, Gautam Kumar, and Aditya Akella. Congestion control in machine learning clusters. In *HotNets 2022*, pages 235–242. ACM, 2022.

[73] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *OSDI 2020*, pages 463–479. USENIX Association, 2020.

A Background of OrbWeaver

OrbWeaver [21] is a framework designed for the opportunistic transmission of data within modern programmable networks. As detailed in its paper, OrbWeaver imposes a negligible impact on user traffic, the computational/state resources of participating switches, or their power consumption.

Instead of "measuring" the utilization of each port, OrbWeaver continually generates a stream of low-priority packets. Leveraging the behavior of priority queues, when there are no high-priority user packets in the queue, the low-priority packets produced by OrbWeaver can effectively utilize the available bandwidth and are transmitted to neighboring switches. Essentially, OrbWeaver fills these bandwidth gaps based on the characteristics of the priority queue.

OrbWeaver also supports the amplification (*i.e.*, multicast) of low-priority packets, as demonstrated in its paper. In InvisiFlow, we require only one multicast group to multicast pull requests to all neighboring switches. For Intel Tofino switches, the multicast group ID can be configured in the control plane, enabling the data plane to utilize this ID for multicasting packets.

B Workloads

Our evaluation workloads are configured as follows.

Hadoop workload (FB_Hadoop). By default, we adopt the FB_Hadoop workload [40]. In this workload, around 60% of the flow sizes are smaller than 1 KB, and the flows arrive according to a Poisson process as in previous work [17, 19].

Machine learning workload (ML). We emulate an ML workload to demonstrate InvisiFlow's performance under different traffic patterns and distributions. In this workload, all flows are large (1 MB each), and multiple ML jobs are running concurrently. We randomly select eight servers for each ML job and these servers periodically exchange data with each other, as described in [72, 73]. The duration of the period is calculated based on the load rate.

C Flow Error for Heavy Hitters

This section shows the normalized relative error of the flow size for heavy hitters with more than 10^3 packets.

The results are similar to the relative error of all flows shown in §6.2. Under the default setting, InvisiFlow at a 70% load exhibits a reduction of approximately 1.6× and 4.9× compared to low-priority UDP and pull-based baselines, respectively. In the asymmetric topology, the normalized relative error of all flows can reach up to 14.4 for low-priority UDP and 54.8 for pull-based baselines.