PPT: A Pragmatic Transport for Datacenters

Lide Suo[1*], Yiren Pang[1*], Wenxin Li[1,2†], Renjie Pei[1], Keqiu Li[1], Xiulong Liu[1], Xin He[1], Yitao Hu[1], Guyue Liu[3]

Tianjin Key Laboratory of Advanced Networking, Tianjin University[1]
Huaxiahaorui Technology (Tianjin) Co., Ltd.[2]
Peking University[3]

ABSTRACT

This paper introduces PPT, a pragmatic transport that achieves comparable performance to proactive transports while maintaining good deployability as reactive transports. Our key idea is to run a low-priority control loop to leverage the available bandwidth left by the reactive transports. The main challenge is to send just enough packets to improve performance without harming the primary control loop. We combine two unconventional techniques: an intermittent loop initialization and an exponential window decrease, enabling us to dynamically identify and fill the spare bandwidth. We further complement PPT's design with a buffer-aware flow scheduling scheme to optimize the average FCT of small flows without prior knowledge of flow size information. We have implemented a PPT prototype in the Linux kernel with ~400 lines of code and demonstrated that compared to Homa, it delivers up to 46.3% lower overall average FCT and even 25%/55.5% lower average/tail FCT of small flows in an Memcached workload.

CCS CONCEPTS

• **Networks** → **Transport protocols**; **Data center networks**.

KEYWORDS

Data center networks; pragmatic transport; dual-loop rate control; flow scheduling

ACM Reference Format:
Lide Suo[1*], Yiren Pang[1*], Wenxin Li[1,2†], Renjie Pei[1], Keqiu Li[1], Xiulong Liu[1], Xin He[1], Yitao Hu[1], Guyue Liu[3]. 2024. PPT: A Pragmatic Transport for Datacenters. In *ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3651890.3672235

1 INTRODUCTION

With the link speed growing from 1/10G to 100G and beyond, many datacenter network (DCN) transport protocols have been proposed to provide ultra-low latency for applications. These protocols can be divided into two categories. One is reactive transport (e.g., DCTCP [5], TIMELY [29]) that usually starts from a small initial window and then iteratively ramps up to the right rate. Another is proactive transport (e.g., NDP [15], ExpressPass [11], Homa [32], Aeolus [17]), which allocates bottleneck link bandwidth as credits to senders who can then send packets at optimal rates.

Despite many of them, current solutions either exhibit low performance or bring practical issues (§2.1). On the one hand, reactive transports are cautiously designed and are hard to reach the right rate in each round, thus leading to under-utilized networks. On the other hand, proactive transports may be promising for delivering high performance but are hard to deploy because (i) they require a non-trivial amount of labor to re-implement their transports in the Linux kernel and even significant code modifications to applications; (ii) they assume a prior knowledge of flow size for scheduling, which is unavailable in practice.

In this work, we aim to explore a new transport that can achieve comparable performance to proactive transports while being readily deployable. *We start with the reactive transports to leverage their broad deployment advantage and ask whether it is possible to address the performance issues arising from their reactive nature.*

Answering this question requires first addressing the under-utilization issue of current reactive transport. Indeed, our preliminary experiments reveal that if the spare bandwidth can be ideally utilized at each round, DCTCP even achieves a ~33% lower overall average flow completion time (FCT) than Homa [32] (§2.3). While promising, realizing such idealized utilization encounters two concrete challenges. How do we quickly detect when DCTCP does not use the available bandwidth? How to gracefully use the available bandwidth without harming the primary DCTCP traffic.

One plausible way could run the low-priority loop proposed by prior work RC3 [30] in parallel with the normal DCTCP loop to utilize the spare bandwidth. Unfortunately, RC3 lets the low-priority control loop fill up the entire BDP (bandwidth-delay product) for every RTT, thus occupying too much switch buffer and affecting the injection of normal packets. As a result, it may even perform worse than the original DCTCP, with the average FCT of small flows prolonged by 1.87× in some workload (§6).

We address these challenges with PPT, a pragmatic DCN transport that aims to retain the deployability advantages of DCTCP while enabling it to efficiently utilize the available bandwidth to optimize FCTs. PPT relies on two parallel control loops: one high-priority control loop (HCP) transmitting normal DCTCP packets and a second low-priority control loop (LCP) transmitting opportunistic packets to fill the pipe.

[*]Both authors contributed equally to this research.
[†]Corresponding author.

ACM SIGCOMM '24, August 4–8, 2024, Sydney, NSW, Australia
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0614-1/24/08.
https://doi.org/10.1145/3651890.3672235

HCP is exactly DCTCP, while LCP is based on a key insight that filling the gap of the congestion window to the historical observed maximum value for each flow in each round could help to gracefully utilize the spare bandwidth left over by DCTCP (§2.3). To this end, LCP exploits two unusual techniques. First, it employs an *intermittent loop initializing* mechanism (§3.1) to identify the spare network capacity, which initializes an LCP loop when a flow starts or exhibits the lowest queue occupation in its HCP loop. Second, it revolves around an *exponential window decreasing (EWD)* strategy (§3.2) to ensure the opportunistic packet transmissions can utilize the spare bandwidth gracefully, i.e., without causing bandwidth waste and bursting normal HCP packets.

This dual-loop rate control design is effective for three reasons. *First*, it intermittently initializes and terminates LCP to match the dynamically emerged spare bandwidth left by HCP. *Second*, with the EWD strategy, LCP ensures its window plus the current HCP's one does not exceed the maximum window for each flow in every RTT. *Third*, it uses ECN for LCP loop to make it sense congestion and decrease the sending rate early before buffer overflow, so as to protect normal HCP packets. These make PPT efficiently utilize the available bandwidth and hence reduce overall FCTs.

Despite being effective, the dual-loop rate control treats small and large flows equally in the network. Thus, the packets of small flows may be queued after those of large flows in the switch buffer. So, we must also tackle concerns about the average FCT of small flows. Indeed, flow scheduling is effective for this due to its use of in-network priorities. Nevertheless, designing an effective flow scheduling scheme while retaining PPT's pragmatic merit needs to overcome one critical challenge: How do we schedule flows without prior knowledge of flow size and without affecting the benefits of the dual-loop rate control on overall FCT reduction?

To address this challenge, we present a buffer-aware flow scheduling scheme (§4). It first exploits a flow identification approach to identify large flows by monitoring the data size copied into the TCP send buffer through the first system call. Then, it uses a mirror-symmetric packet tagging method to assign priorities. Precisely, it evenly divides the available priorities into a high-priority part for HCP and a low-priority part for LCP. Each part assigns its lowest priority to identified large flows, and gives the highest priority to non-identified flows in the beginning and then gradually decreases their priorities as they send more data. Through this way, PPT ensures that small flows can bypass queued packets of most large flows in the network from the start of transmission, while guaranteeing HCP will not get harmed by LCP.

We have implemented a PPT prototype that requires only end-host implementation (§5). Specifically, we implement the LCP control logic and the scheduling process as Linux kernel modules that sit within the TCP/IP layers. Further, we make minimal modifications to the kernel source code to avoid LCP impacting HCP. While the implementation digs shallowly into the kernel code, it does not impact PPT's compatibility with legacy TCP/IP network stacks.

We further built a small-scale testbed in the CloudLab cluster, together with large-scale trace-driven simulations at 40/100Gbps, to evaluate the performance of PPT (§6). We have the following key findings:

- Compared to the reactive transport—RC3 [30], PPT reduces the overall average FCT and the average/tail FCT of small flows by up to 92.7% and 99.2%/99.9%, respectively. Moreover, the performance advantage of PPT over RC3 still holds even when we limit the available switch buffer for RC3's low-priority queues.
- PPT achieves lower overall average FCT than the proactive transports, NDP [15], Aeolus [17], and Homa [32], under both oversubscribed and non-oversubscribed topologies. Compared to Homa, it reduces the overall average FCT by up to 46.3% and even shows a 25%/55.5% lower average/tail FCT of small flows in an Memcached workload.
- PPT's design components are effective for the performance, and it is robust to parameter settings like TCP send buffer and ECN marking threshold. Moreover, PPT can also outperform PIAS [9] and HPCC [25] and its design can be integrated with delay-based transport.

2 BACKGROUND AND MOTIVATION

2.1 Drawbacks of Existing Transports

We first provide a detailed motivation of our work by highlighting the drawbacks of existing transports (see Table 1).

Limitations of reactive transports: Reactive transports are essentially sender-driven and most of them can work with commodity switches and retain compatibility with TCP/IP stacks, yet without any modifications to applications. However, they fall short in utilizing spare bandwidth or optimizing FCTs. For example, solutions like DCTCP [5] and PIAS [9] are TCP-style transports that may waste bandwidth in the slow-start stage. TCP-10 [12] and Halfback [23] try to utilize spare bandwidth in the startup phase by increasing the initial window to 10 or pacing out <141KB small flows in 1st RTT while ignoring those in the queue buildup phase. RC3 [30] uses a second control loop to utilize the spare bandwidth left by the primary TCP loop; however, it sends too excessive packets and adversely impacts the performance. HPCC [25] can gracefully utilize spare bandwidth but require precise INT feedback from switches. In summary, reactive transports except PIAS [9] mainly perform rate control while do not take advantage of in-network priorities.

Limitations of proactive transports: Another line of work attempts to re-architect reactive TCP-style transports by *proactively* allocate bottleneck link bandwidth, known as proactive transports. They typically allocate bandwidth as credits to active senders who then can send at right rates. Nevertheless, they have the following shortcomings. First, they suffer from a pre-credit dilemma. Meaning, their senders either passively hold packets before receiving credits (e.g., [11, 15]), causing under-utilization in the 1st RTT, or aggressively injecting packets at line rate for all flows during pre-credit phase (e.g., [17, 32]), causing bursting. Second, some of them like Homa [32] and Aeolus [17] assume a prior knowledge of flow size for scheduling. Third, they require refactoring the TCP/IP stack to implement their transport logic. For example, Homa-Linux [33] involves ~10k lines of code for many redesigned network subsystems or even 42.2% of the application code to be modified (appendix C).

2.2 Desirable DCN Transport Properties

To take a leap forward and design a pragmatic datacenter transport, we list 3 key desirable properties:

Schemes		Efficiency		Practicality		
		Spare bandwidth utilizing pattern	Scheduling without flow size	Working with commodity switches	Compatible with TCP/IP stacks	Non-intrusion to applications
Reactive	DCTCP [5]	Passive	×	Yes	Yes	Yes
	TCP-10 [12]	Passive	×	Yes	Yes	Yes
	Halfback [23]	Passive	×	Yes	Yes	Yes
	RC3 [30]	Aggressive	×	Yes	Yes	Yes
	PIAS [9]	Passive	Yes	Yes	Yes	Yes
	HPCC [25]	Graceful (but INT required)	×	No	No (for RoCE)	Yes
Proactive	Homa [32]	Aggressive	No (flow size required)	Yes	No	No
	Aeolus [17]	Aggressive	No (flow size required)	Yes	No	No
	ExpressPass [11]	Passive (1st RTT wasted)	×	Yes	No	No
	NDP [15]	Passive (1st RTT wasted)	×	No	No	No
PPT		**Graceful**	**Yes**	**Yes**	**Yes**	**Yes**
× denotes the relevant schemes focus only on controlling flow sending rates, while do not use in-network priorities for flow scheduling.						

Table 1: Summary of prior transports in literature and comparison to PPT.

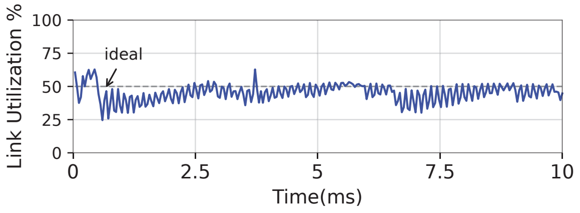


Figure 1: Link utilization of DCTCP under 0.5 load.

Utilizing spare bandwidth gracefully: Modern applications like large-scale deep learning on GPU, storage on NVMe, have driven the datacenter link speed to grow from 1Gbps to 100Gbps and beyond. Hence, it is important to fully utilize available bandwidth in every single RTT. Meaning, the transport solution should sent *just enough packets* to gracefully utilize the spare bandwidth.
Scheduling flows without flow size: Another application requirement is low latency, especially for very short flows (≤1KB) that dominate a high volume in the workload. Flow scheduling is effective for achieving low latency, which uses in-network priorities to prioritize small flows over large ones. A pragmatic transport should integrate this technique to effectively schedule flows in a real-world setting where flow size is not known a prior [9].
Being readily-deployable: Finally, a pragmatic transport solution must work with existing commodity switches and should be backward compatible with legacy TCP/IP network stacks. From the perspective of usability, the transport must also incur no modifications to application code.

2.3 The Design Space

PPT starts with the reactive transport DCTCP to leverage its practical advantages while remedying its inefficiencies in utilizing the spare bandwidth and optimizing FCTs.
DCTCP is a good start. DCTCP [5], though proposed more than 10 years ago, is an appropriate fit for starting a pragmatic transport design for two reasons. First, DCTCP is widely adopted in industry [19, 35] and has been integrated into various OS kernels [2, 3]. Second, it requires no switch modification and is application-friendly. Despite this, DCTCP still leaves a significant performance gap.
The key culprit is under-utilization. In DCTCP, spare bandwidth emerges in two cases. *The first one is in the first few RTTs.* A new DCTCP flow starts with a small initial window and usually takes several RTTs to ramp up the congestion window to reach the maximum rate [12] [26]. *Another case is in the queue buildup phase.* To show this point, we run an ns-3 simulation with two senders and one receiver. The bottleneck link capacity is 40Gpbs. The ECN marking threshold is 120KB. Flows are generated based on the Web Search workload [34] and follow Poisson arrival process. We sample the bottleneck link utilization every 100us for 10ms when DCTCP enters a steady state. As we set the network load to 0.5, the ideal link utilization should be maintained at 50%. However, as we can see from Fig. 1, the link utilization achieved by DCTCP fluctuates between 25% and 50% at most of the time.
Remarks: The intuitive reasons are as follows. DCTCP marks ECN at the switch for arriving packets if queue occupancy exceeds a threshold K, and the sender cuts the window based on the fraction of ECN marked ACKs. So, when queue buildup forms, there are generally two statuses for the active flows. First, multiple flows are marked with ECN and cut their windows simultaneously, thus causing a sudden drain on the switch buffer. This is why the link utilization in Fig. 1 can drop to 25% in a specific time, leaving half of the bandwidth under-utilized. Second, only one flow perceives this congestion while others are still probing their maximum congestion window. Therefore, when this flow cuts its window, the buffer occupancy may drain to a relatively low value. In this case, there is also some spare bandwidth, thus making the link utilization fall below 50% at most of the time.
Filling the gap of DCTCP sounds good. To show the potential value of filling DCTCP's gap, we consider a *hypothetical* DCTCP. It was constructed in the following. We first run the default DCTCP and record each flow's maximum window (MW). Then, we run the *hypothetical* DCTCP that sends just enough opportunistic packets to fill the gap to MW for each flow in each RTT. We simulate a leaf-spine topology consisting of 144 servers, 9 leaf switches and 4 spine switches, with the edge and core links being operated at 40Gbps and 100Gbps, respectively. The flows are generated following an all-to-all traffic pattern, again with the Web Search [34] workload, at 0.5 network load. The results are shown in Fig. 2. We observe that the hypothetical DCTCP reduces the overall average FCT by 33%[1] and 40%, compared to Homa and NDP, respectively. The reason is that the hypothetical DCTCP sends just enough opportunistic packets to utilize the leftover bandwidth. In contrast, both Homa

[1]The hypothetical DCTCP does not contain flow scheduling component, thus its improvement over Homa is smaller than that of PPT (§6.2).

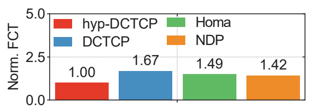


Figure 2: Overall avg. FCT under different schemes.

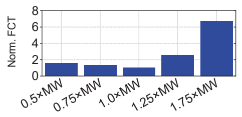


Figure 3: Filling gap to different window.

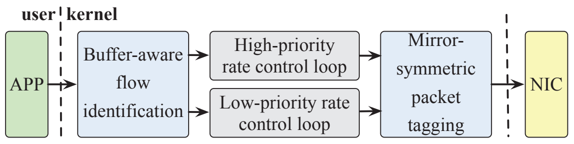

user
kernel
APP
Buffer-aware flow identification
High-priority rate control loop
Low-priority rate control loop
Mirror-symmetric packet tagging
NIC

Figure 4: PPT overview.

and NDP send unscheduled data packets at line rate in the pre-credit phase (i.e., the phase before receiving credits), thus incurring too many unnecessary packet losses and retransmissions.

Filling the gap to MW is a good choice. Filling fewer opportunistic packets wastes link capacity, while filling too many impacts normal DCTCP packet transmissions. We advocate filling the gap to the MW for each flow. To show why this works, we run a simulation with the same topology as above and use a Data Mining workload [13]. The network load is 0.6. We use different window sizes to be filled by the opportunistic packets, varying from 50% to 150% of the MW. Fig. 3 shows the results. We have two findings. First, if the window value that needs to be filled is smaller than MW (e.g., 50%×MW), the performance degrades a lot (e.g., 56% higher FCT). Second, if this value is larger than MW, the sender sends a burst of opportunistic packets in each round, resulting in a significantly high loss rate, with the FCT increased by up to 6×. As a result, we consider filling the gap to 1×MW is a good choice.

Scheduling flows with in-network priorities is a must. While filling the gap of DCTCP could reduce overall FCTs, small flow's FCT optimization is missing. An effective way is to schedule flows with in-network priorities, e.g., assigning higher priorities for small flows to bypass queued packets of large flows in the network. However, we need to address two issues. First, how to differentiate large flows from small ones on the premise that flow size is not known a prior? Second, how to prioritize small flows while incurring no penalty on overall FCTs or even without starving large flows. For the first issue, PIAS [9] uses bytes sent to differentiate large flows gradually, but it is too late to isolate small flows from large ones. QCLIMB [24] uses learned lower bounds for differentiating flows from the beginning of transmission but requires a daunting machine learning model. Our design advocates a buffer-aware approach to differentiate a majority (86.7% as revealed in §4.1) of large flows at the beginning of transmission and differentiate the remaining during transmission with a flow-aging idea like PIAS [9]. For the second issue, we take a mirror-symmetric packet tagging method to prioritize small flows over large ones (§4.2).

Putting it all together. PPT is a software only transport solution running within end-hosts that requires no modifications to existing NICs, switches and even applications. It mainly contains the following two components (Fig. 4):

- **Dual-loop rate control**[2] **(§3):** PPT has a **h**igh-priority **c**ontrol loo**p** (**HCP**) and a **l**ow-priority **c**ontrol loo**p** (**LCP**). HCP uses the default DCTCP congestion control algorithm [5] to transmit normal packets. LCP revolves around HCP and aims to send opportunistic packets to gracefully utilize the spare bandwidth left by the primary HCP loop.
- **Buffer-aware flow scheduling (§4):** To achieve the lowest possible latency, PPT integrates a flow scheduling component with the dual-loop rate control. It first uses a buffer-aware approach to identify large flows. Then, it leverages a mirror-symmetric packet tagging method to prioritize small flows over large ones in each rate control loop.

We want to highlight that PPT's two components work organically. When a flow starts, it first goes through the dual-loop rate control, where the HCP sends normal packets in order from the first byte in the send buffer while LCP sends opportunistic packets from the very last byte. This flow will further go through the buffer-aware flow scheduling component, where its normal packets use the first four high priorities while the opportunistic packets use the remaining four low priories (see §4.2 for the details of priority tagging). So, PPT is not simply dividing the traffic into multiple classes and coupling flows with different classes or priorities to utilize the spare bandwidth. Instead, it divides each flow into two parts and uses the low-priority part to fill the bandwidth left by the high-priority part.

3 DUAL-LOOP RATE CONTROL

Fig. 5 illustrates the dual-loop rate control logic of PPT. As we can see from this figure, HCP is simply the same with DCTCP [5]; thus, we omit its details. LCP sends opportunistic packets from the tail end with the following two techniques:

- **Intermittent loop initialization (§3.1):** Because of the window dynamics nature in DCTCP, the spare capacity left by the HCP loop appears intermittently. Whenever this occurs, PPT opens an LCP loop and calculates an appropriate initial window for opportunistic packet transmissions.
- **Exponential window decreasing (§3.2):** In each LCP loop, PPT introduces a novel exponential window decreasing mechanism, which ensures, in every RTT, the window summation of LCP and HCP will not exceed the MW a flow experienced in the past. As such, LCP can gracefully utilize the spare bandwidth left by HCP.

Remarks: RC3 [30] also uses low-priority control loop to fill the spare bandwidth left by TCP-like transport. However, we want to highlight that PPT is designed with radically different LCP rate control, in terms of *when, how and what. First*, PPT intermittently detects when the spare bandwidth appears and initializes or terminates the LCP accordingly, while RC3 keeps the LCP loop opened until it crosses with HCP loop's traffic. *Second*, PPT uses ECN for LCP loop to make it sense congestion early and decreases the sending rate to not affect HCP loop, while RC3 makes no effort to protect HCP loop's transmission. *Third*, PPT uses the EWD window decreasing strategy to send a suitable amount of opportunistic packets in every RTT to gracefully utilize the spare bandwidth without incurring buffer overflow. Worse still, RC3 uses in-network priorities

[2]Note that in the Internet context other than DCN, PCC Protues [28] is the closest work to PPT, which uses the traffic of low-priority applications to fill the bandwidth leftover by high-priority applications. However, PPT uses different bandwidth padding scheme. That said, it uses the opportunistic packets starting from the end of the same flow to fill the spare bandwidth. Moreover, PCC Protues [28] requires application to specify its priorities, while PPT gradually demotes a flow's priority as more bytes sent.

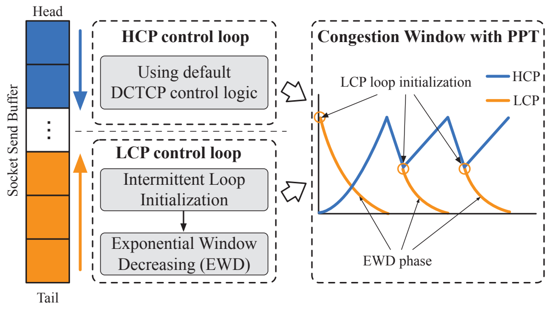

Head
Socket Send Buffer
Tail
HCP control loop
Using default DCTCP control logic
LCP control loop
Intermittent Loop Initialization
Exponential Window Decreasing (EWD)
Congestion Window with PPT
LCP loop initialization
HCP
LCP
EWD phase

Figure 5: Dual-loop rate control illustration.

only for prioritizing HCP traffic, making no attempt to prioritize small flows over large ones (i.e., lacking flow scheduling). These make RC3 shows worse performance as compared to PPT (see §6).

3.1 Intermittent Loop Initialization.

Typically, for a DCTCP flow, its congestion window size exponentially increases in the beginning and then exhibits a well-known "sawtooth" behavior. Meaning, the spare bandwidth left by HCP appears intermittently and mainly in the following two cases.

Case 1: spare bandwidth in the first few RTTs.
Since HCP uses DCTCP, a new flow typically needs to go through the slow-start and congestion-avoidance phases, which are the common features in TCP and take multiple RTTs to ramp-up to the maximum window (MW). So, before the flow's window reaches the MW, there exists significant spare capacity left over by HCP.

Case 2: spare bandwidth in queue build-up phase.
After the startup phase, a DCTCP flow's congestion window exhibits a sawtooth behavior. During this period, the queue length grows until packets are dropped, similarly evolving in the sawtooth pattern. Specifically, when the queue size exceeds the marking threshold K, the sources will receive ECN marks in one RTT later and cut their windows accordingly to lower the queue length. When multiple concurrent flows cut windows simultaneously, the queue length may decrease to a very low value or even zero, thus leaving some spare bandwidth either.

Note that the rise of high-speed DCN and the heavy-tailed distribution of DCN workload enable many small flows to complete within the first RTT [17]. When every new flow opens an LCP loop directly at its start, large flows will compete for bandwidth in the 1st RTT, thus adversely affecting small flows' FCTs. To mitigate such impact, PPT, in case 1, will only open LCP loops directly in the 1st RTT for normal flows but delay the LCP loop initialization to the 2nd RTT for those identified as large by the buffer-aware approach (§3). For spare bandwidth emerging in case 2, PPT will directly open an LCP loop for each flow.

The remaining task is to decide an initial congestion window for each LCP loop, as detailed below.

LCP loop initialization in case 1: Ideally, the initial window of the LCP is determined by the spare bandwidth left by the HCP. However, in case 1, it is hard to know how much spare bandwidth is left since the DCTCP flow itself is in the start-up phase to probe the available bandwidth in the network. Therefore, in this case, we set the initial congestion window I for the opportunistic packet in the LCP loop to the bandwidth-delay product (BDP) minus the initial window of the relevant DCTCP flow.

LCP loop initialization in case 2: The spare bandwidth in case 2 is closely related to the queue length dynamics. Meanwhile, we observe that the evolving trend of the queue length can be expressed by DCTCP parameters. More precisely, DCTCP uses a parameter α at the sender to estimate the fraction of packets that are marked with ECN, which is updated every RTT as follows:

$$\alpha \leftarrow (1-g) \times \alpha + g \times F \tag{1}$$

where F represents the ratio of ECN marked packets in the past window of data, and $0 < g < 1$ is a weight factor used to emphasize how much weight F should be given when estimating α. Note that the DCTCP sender receives ECN marks for every packet when the queue length exceeds the marking threshold K and does not receive any marks when the queue length is below K.

Essentially, α provides the relation between the queue size and marking threshold K – with α close to 0 and 1 indicating low and high levels of congestion, respectively. A key implication behind the definition of α is that the smaller the parameter α, the higher the probability that the network has spare capacity. Therefore, after a flow is over the startup phase in HCP loop, PPT initializes an loop for it whenever its DCTCP parameter α takes the minimum value α_{min} in the past RTTs. Also, the initial congestion window is calculated as follows:

$$I = (\frac{1}{2} - \alpha_{min}) \cdot W_{max} \tag{2}$$

Here, the term W_{max} is the maximum value of the congestion windows[3] the flow experienced in past RTTs. The rationales for Equation (2) are two-fold. First, the larger the α_{min}, the less volume of opportunistic packets should be injected into the network, thus the smaller initial window I should be configured. Second, because DCTCP will cut its window by half at most when congestion is perceived, Equation (2) ensures that the initial window of opportunistic packets is at most half of the MW.

3.2 Exponential Window Decreasing

As mentioned in §3.1, the spare bandwidth left by the HCP loop appears in two cases. In case 1, the new flow first goes through the slow-start phase and enters the congestion-avoidance phase to increase the window additively. In case 2, the flow just reacts to severe congestion and starts to additively increase its window again. In both cases, the spare bandwidth keeps decreasing. Hence, we let the LCP loop to exponentially decrease its congestion window, to gracefully utilize the spare bandwidth. To this end, we cut the sending rate of opportunistic packets by half every RTT. This way can smoothly utilize the spare bandwidth, without impacting normal packet transmission, while avoiding underutilization.

We combine the sender and receiver to enforce this EWD strategy. At the beginning of an LCP loop, the sender paces the initial sending rate of opportunistic packets to the initial window divided by RTT, i.e., I/RTT. Each opportunistic packet enters the low-priority queue in the switch port and will be marked with ECN if buffer occupancy is greater than a predefined threshold (will discuss later) upon its arrival. LCP rate control is implemented on the receiver side. In

[3]Note that when two large flows arrive one after the other and share a common bottleneck link, the latter flow may record a lower value of W_{max} and thus send fewer opportunistic packets via LCP than the earlier one. This is indeed an unfair issue, but it may not matter much. This is because only the congestion windows after the congestion-avoidance phase are considered for computing W_{max}.

particular, whenever two consecutive opportunistic data packets arrive at the receiver, the receiver sends one low-priority ACK back to the sender. Upon receiving a low-priority ACK, the sender sends an opportunistic data packet. Since receivers only generate one low-priority ACK for receiving two opportunistic packets, ideally, the sending rate of opportunistic packets is naturally cut by half after every single RTT.

Remarks: We now highlight two remarks about the design. *First*, PPT uses the ECN marking to enforce the LCP loop's opportunistic packets not to impact the HCP loop's normal packets. If a low-priority ECN-marked ACK arrives at the sender, it indicates two possible situations. The first situation is that normal packets block opportunistic ones. If LCP keeps the sending rate of opportunistic packets, it will increase the queuing delay or even packet loss. The second situation is that opportunistic packets impact normal packets. Though we isolate them into different priority queues, all data packets essentially share the switch buffer. So, when the sender receives a low-priority ECN-marked ACK, it ignores this ACK and does not trigger new opportunistic packet, to preserve the normal packet transmission. *Second*, each LCP loop will not last forever. Our design will terminate the current LCP loop if the sender has not received low-priority ACKs for 2 RTTs and start to discover spare bandwidth again to initialize a new LCP loop.

ECN-marking threshold: The choice of ECN marking threshold is important as it directly affects the tradeoff between throughput and latency [5, 39, 40]. As the most instantaneous ECN-based datacenter congestion control algorithms [5, 10, 39, 40] suggested, we re-use the RED marking scheme implemented commodity switches and simply set both the low and high thresholds to the same value K. The ideal marking threshold K is calculated as follows:

$$K = \lambda \cdot C \cdot RTT \tag{3}$$

where C and RTT are the link speed and the base round-trip time, respectively, and we assume they are fixed values of the network. So, the marking threshold K is only changing with parameter λ.

For the high-priority queue, λ is chosen as the same as the DCTCP paper (0.17 in theory [31]). For the low-priority queue, if the value of K is too large, the opportunistic packets may be queued, thus hurting normal packets. Meanwhile, a too-small value of K may lead to the premature marking of opportunistic packets and leave spare bandwidth underutilized. Considering this tradeoff, PPT sets a slightly smaller λ (i.e., 0.1 by default in this paper) for the low-priority queue. We also show in §6.3 that PPT has performance benefits under a wide range of λ for the low-priority queue.

4 BUFFER-AWARE FLOW SCHEDULING

4.1 Buffer-aware Flow Identification

A flow's data is first generated by an application. Then, the application invokes system call functions to copy the data into send buffer in the OS kernel. Finally, the NIC transmits the data from the send buffer into the network.

We mainly consider the typical case where network transmission rate is smaller than the data generation rate[4], and the data that cannot immediately get transmitted to the network will be queued in TCP send buffer. This has two possibilities. *First*, if the send buffer room is not enough, the application's data copy process will be blocked, thus pushing buffer pressure back to application. *Second*, if there are adequate send buffer, data copy will not get blocked and data will be buffered in send buffer.

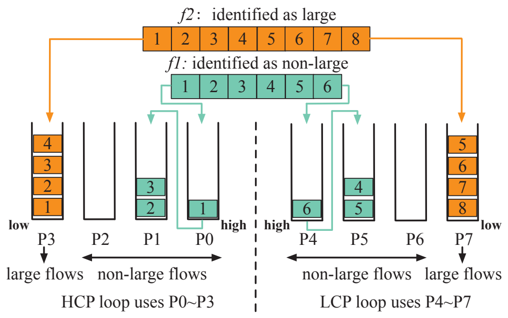


Figure 6: Illustration of mirror-symmetric packet tagging (Note here P0>P1· · · >P7).

Based on the analysis above, we are motivated to configure a relatively large TCP send buffer to enable as much as possible data stored in the send buffer. In this way, we can have a chance to differentiate large flows. That said, we can check the send buffer in the OS kernel at the start of each flow and identify a flow as a large one if its first system call injects an amount of data exceeding a given threshold into the send buffer.

To validate the effectiveness of this method, we run two applications, Memcached and Web Server, using two hosts connected via a 25Gbps network. For both applications, we use one host as client and another one as server. The Memcached application uses the ETC trace of the paper [8], while the web server application follows the Youtube HTTP trace [18]. We set the TCP send buffer to the default value—16KB, which is large enough for both applications. The base RTT is 100μs. The identification threshold is set to 1KB and 10KB, for the Memcached and web server applications, respectively. We observe that for the Memcached application, there are 4098 >1KB flows, and the buffer-aware approach can accurately identify 86.7% of them as >1KB via monitoring the amount of data injected into the send buffer by the first system call. For the web server application, the accuracy of identifying >10KB flows can also reach 84.3%.

Note here, the buffer-aware approach may still leave some portion of unidentified large flows. For these flows, PPT fall-back to PIAS [9] and gradually identify them in transmission stages. Precisely, it uses the bytes sent to estimate its pending data. The more bytes a flow transferred, the more likely it could be a large flow, and accordingly the lower priority it would get in the network.

4.2 Mirror-symmetric Packet Tagging

Packet tagging is performed at end-hosts, which inserts a priority value in each packet's header. With these tags, switches can use strict priority queuing to dequeue packets. To be effective for PPT's design, it should assign higher priorities for flows with fewer bytes to reduce the average FCT of small flows, while not affecting large flows and not making HCP traffic harmed by LCP.

[4]When transmission rate is larger than or equal to date generation rate, very little data will be backlogged in the send buffer. In this case, network is not the bottleneck for the application and any transport design would be to paint the lily. So, we ignore this case as it is not the norm in datacenters with modern bandwidth-intensive applications like distributed machine learning and cloud storage [38].

Hence, we use a mirror-symmetric packet tagging method. As shown in Fig 6, it divides the priorities into two parts. The first high-priority part (i.e., P0∼P3) is used for HCP traffic, while the second low-priority part (i.e., P4∼P7) is for LCP traffic. Each part uses the same mechanism to assign priorities, where the flows identified as large by the buffer-aware approach use the last lowest priority, and the unidentified flows use the remaining three high priorities. Thad said, if a flow is identified as a large one, the sender allocates the lowest priority P3 in the first part to its HCP packets and the lowest priority P7 in the second part to the LCP packets. Otherwise, the flow's HCP and LCP packets will be tagged with P0 and P4, respectively. As more bytes are sent, the HCP and LCP packets will be tagged at the same pace with decreasing priorities P_i and P_{i+4} ($1 \leq i \leq 3$), respectively.

4.3 Why this Works

The effectiveness of buffer-aware scheduling is two-fold. First, its buffer-aware approach can identify most large flows, enabling the packet tagging method to assign large flows with lower priorities from the start of transmission and reducing the risk of small and large flows coexisting in high priority levels. Second, it separates the HCP and LCP traffic into different priorities. This not only ensures LCP will not harm HCP but also guarantees large flows will not get starved (as the HCP packets of identified large flows have higher priority than the LCP packets of unidentified flows). Therefore, our PPT can achieve lower overall average FCTs than the state-of-the-art transport—Homa [32] (§6).

For small flows, PPT can achieve comparable or even better performance than Homa, because Homa blindly transmits unscheduled packets at line rate in the 1st RTT. But this depends on the workloads. For workloads entirely composed of small flows (e.g., Memcached workload [4]), most flows can be totally sent out in the 1st RTT. This could increase network load unpredictably and cause burst or even packet losses, making Homa achieve higher FCTs of small flows than PPT. For workloads with polarized flow sizes, e.g., data mining [13] containing both tiny flows (<1KB) and massive flows (<100MB), Homa will not be affected too much by the 1st RTT issue and is slightly better than PPT for small flow performance. For typical heavy-tailed workloads like web search [34] with more diversified flow sizes, PPT achieves comparable performance with Homa for small flows.

5 IMPLEMENTATION

We have implemented the PPT prototype as an extension to the Linux Kernel 3.18.1 with ∼400 lines of code. Our implementation sits within the TCP/IP layer. As shown in Fig. 7, we make minimal modifications to the kernel network stack to implement the PPT control logic. Because we do not touch the core congestion control code, PPT is architecturally compatible with legacy TCP/IP stacks. Note that PPT only needs to configure ECN marking and strict priority queuing (SP) at switch side, which are standard features in all existing commodity switches.

5.1 Sender

In the original Linux kernel, the sending pipeline starts data transmissions when an application calls the *send()* function. Then, it

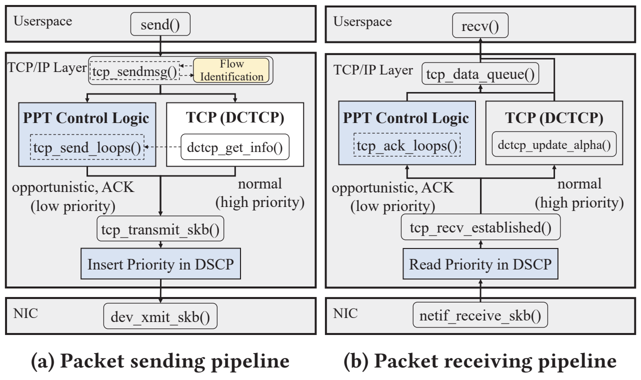


Figure 7: PPT Linux kernel implementation.

copies the application's data from userspace to the kernel send buffer; later on, it invokes the *tcp_sendmsg()* function to segment the buffered data into packets represented by socket buffer (*skb*) data structure. Here, each *skb* represents one packet. The *skb*s will first go through the core TCP logic and then be delivered to the NIC device queue for transmission via the *dev_queue_xmit()* function. PPT extends the sender-side code in the kernel networking stack mainly by adding a new function called *tcp_send_loops()*.

Flow identification: We identify a flow by its 5-tuple: src/dst IPs, src/dst ports, and protocol id. We construct a data structure *msg_iov* in *tcp_sendmsg()*, where the *msg_iov->iov_len* field records the first system call size. If this field's value is larger than a given threshold, we identify the flow as large. We also use the *msg_iov* structure to store the bytes sent for the follow-up packet tagging.

LCP loop invoking: We use *tcp_send_loops()* to invoke the LCP loop once the data has been segmented into *skb*s by *tcp_sendmsg()*. Specifically, the *tcp_send_loops()* function reads the TCP write queue from the tail end to send opportunistic packets. For normal DCTCP packet sending, it still goes through the existing *tcp_push()* kernel function, which reads from the first byte of the TCP write queue. In the *tcp_send_loops()* function, we also need to do rate control and mark packet priority.

Rate control: In *tcp_send_loops()*, we only implement the intermittent loop initializing mechanism for PPT's LCP control logic to determine an initial window for opportunistic packets. For exponential window decreasing, we leave it to *tcp_ack_loops()* and receiver control (as will be introduced in §5.2). For new flows, the initial window is BDP. For active flows, we invoke the *dctcp_get_info()* function to acquire the runtime parameter α in DCTCP; we also use the *sock* structure to store the minimal α in the past few RTTs, with which we can calculate the initial window.

Priority tagging: We set the *skb*→*priority* field to 4∼7 for LCP packet and 0 ∼ 3 for HCP packet, based on the packet tagging loic in §4.2. After the priority assignment, we forward out the *skb* packets via the *tcp_transmit_skb()* function. Here, in the IP layer, we use the *ip_queue_xmit()* function to copy the value of *skb*→*priority* to the DSCP priority bits in the IP header, such that the network switches and receivers can classify different types of packets.

In summary, our changes in the sender-side code are lightweight and isolated with the core congestion control logic. Thus, we can easily make other TCP-like congestion control algorithms to run along with our PPT.

5.2 Receiver

Receiving data packets: We make the following changes for the data packet receiving pipeline. First, when each packet is read in off the wire and converted to *skb* in the IP layer, we copy the DSCP priority filed in the IP header to the *skb→priority* field. Second, in the TCP layer, we identify different types of packets according to their *skb→priority* fields via the *tcp_recv_established()* function. Third, we isolate the low-priority and high-priority packets into different control logic to ensure the standard TCP ACKing mechanism and other variables remain unaffected. On the one hand, the high-priority packets go through the default DCTCP control logic and return ACKs as usual. On the other hand, when receiving two consecutive opportunistic data packets, we send an ACK packet back to the sender at low priority. This ACK carries the cumulative acknowledge sequence and SACK tags to indicate which opportunistic packet is received. Note here, PPT requires SACK to be enabled. Further, if any packet is out-of-order, irrespective of the priority, it will be temporarily stored in an *out_of_order_queue* via the unmodified function *tcp_data_queue()*; it will be forwarded up to application layer buffer after the missing packets arrive.

Receiving ACKs: When receiving a window-sized of high-priority ACKs, we invoke the *dctcp_update_alpha()* function to update the runtime parameter α in DCTCP. For receiving a low-priority ACK, we add a new function *tcp_ack_loops()* to handle it, which contains two basic operations. The first one implements the exponential decreasing mechanism (§3.2). Specifically, if the low-priority ACK is not marked with *ECE* flag, we send a new opportunistic packet from the last byte of send buffer; otherwise, we do nothing. The second one invokes the unmodified function *tcp_sacktag_write_queue()* to update the SACK scoreboard, which records the bytes that have been SACKed.

Typically, the received ACK is at a value less than *snd_nxt* which is the sequence number of the last byte of all normal packets. However, in case the DCTCP sender has crossed paths with the LCP loop's traffic, the receiver gets in-order opportunistic packets and returns the ACK carrying a sequence number (i.e., the first byte of all opportunistic packets) larger than *snd_nxt*. To allow TCP to continue as usual in this case, we tweak the ACK processing by advancing the send queue's head and updating the *snd_nxt* with the new ACK's value.

In summary, though digging shallowly into the kernel code, our implementation is lightweight and does not touch the core TCP code; thus, it does not impact the compatibility of PPT with legacy TCP/IP stacks.

6 EVALUATION

We evaluate our PPT through a combination of testbed experiments and large-scale simulations and show that:

- PPT achieves lower FCTs in practice (§6.1).
- PPT works well in large datacenters (§6.2).
- PPT's design components are effective (§6.3).

6.1 Testbed Experiments

Setup: We build a testbed in the CloudLab cluster, which contains 15 hosts connected to a Dell-s4048 switch. Each host is equipped with a 20-core CPU (Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz), 64G memory, and a 10G NIC (Mellonax CX4). The switch runs Dell networking OS with 50MB memory shared by 54 ports. The base RTT is roughly 80μs. We enable SACK and set the RTO_{min} to 10ms, which is a reasonable setting in DCN environment [5]. We provide a summary of the parameter settings for testbed experiments. Please see Table 3 in appendix A.

	Short flows (0-100KB)	Large flows (>100KB)	Overall average size
Web Search	62%	38%	1.6MB
Data Mining	83%	17%	7.41MB

Table 2: Flow size distributions of realistic workloads.

Workloads: We use two widely adopted realistic DCN workloads[5]: web search [34] and data mining [13]. As shown in Table 2, both workloads are heavy-tailed. We generate the traffic by randomly starting flows following the Poisson process and control the interval arrival time of flows to achieve the desired network load. Using these two workloads, we construct a 15-to-15 (§6.1.1) and a 14-to-1 (§6.1.2) traffic patterns to evaluate our PPT.

Comparisons: We compare PPT with DCTCP [5], Homa [32] and RC3 [30] in testbed experiments. For Homa, we use its Linux implementation [33] (denoted as Homa-Linux hereafter) and set the RTTbytes to 50KB[6]. Also, as recommended by Homa paper [32], we configure the degree of overcommitment to the number of scheduled priority levels (i.e., 2). We set the ECN marking threshold to 100KB for DCTCP. For RC3, we set the send buffer to its recommended value—2GB. Note that RC3 is originally designed for Internet and uses TCP for high-priority loop by default. To make a fair comparison in DCN environment, we use DCTCP for RC3's high priority control loop. For our PPT, we set the send buffer to 128KB, but it works well under a wide range of send buffer settings (see §6.3 for more details).

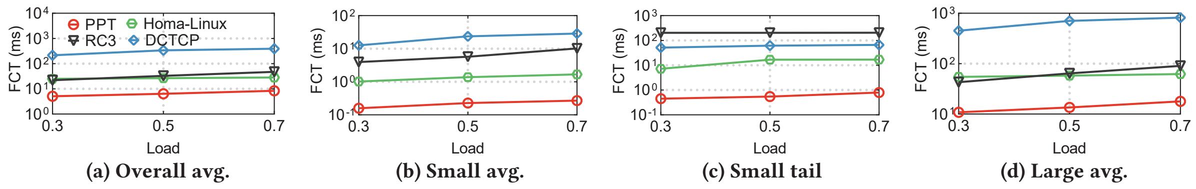
6.1.1 15-to-15 Traffic Pattern.

We construct a 15-to-15 traffic pattern to evaluate our PPT. The results are shown in Fig. 8 and Fig. 9, which present the overall average FCT, the average/tail FCT of (0, 100KB] small flows, and the average FCT of (100KB, ∞) large flows, under the web search and data mining workloads, respectively. From these figures, we make the following observations.

Overall FCT performance: PPT achieves the best performance in the overall average FCT for both workloads. Specifically, compared to Homa-Linux, RC3, and DCTCP, PPT reduces the overall average FCT by up to 79.7%, 82.3%, and 98.1%, respectively, for the web search workload and 28.9%, 17.6%, and 96%, respectively, for the data mining workload. These results verify the effectiveness of PPT in utilizing the available bandwidth in the network.

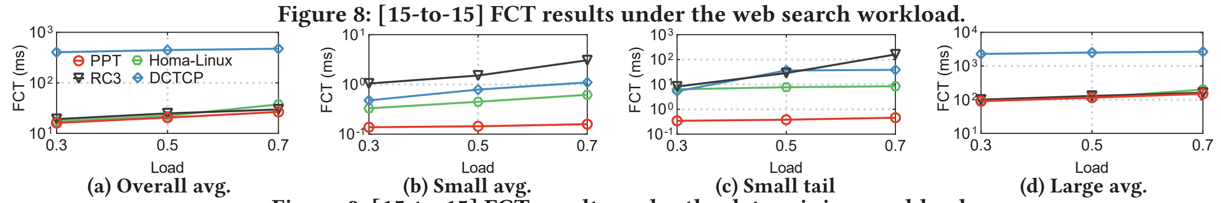
Small flow's performance: PPT achieves significantly better performance for small flows than RC3 and DCTCP. PPT delivers 86.8%~97.4%/95.8%~99.8% and 71.1%~99.1%/93.6% ~99.1% lower average/tail FCT of small flows, respectively, than RC3 and DCTCP, across all tested settings. The poor performance of RC3 and DCTCP

[5]In certain experiments (§6.2), we also use a tiny workload—Facebook memcached that is entirely composed of ≤100KB small flows and was also used in the Homa paper [32].
[6]Note that the 50KB RTTbytes are higher than the 10KB RTTbytes in the original Homa paper [32]. This is because the original Homa is implemented based on DPDK and has extremely lower base RTT (roughly 8μs).



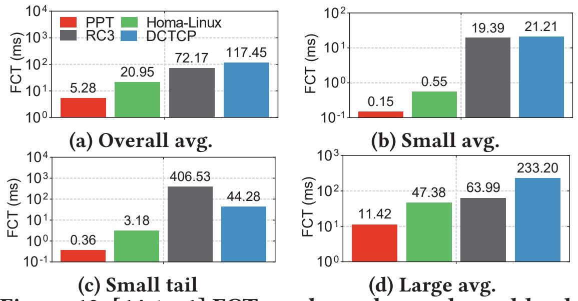
(a) Overall avg. (b) Small avg. (c) Small tail (d) Large avg.

Figure 8: [15-to-15] FCT results under the web search workload.



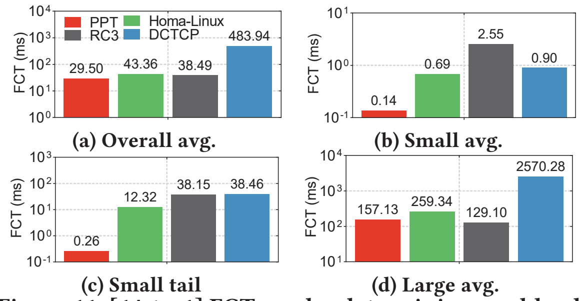
(a) Overall avg. (b) Small avg. (c) Small tail (d) Large avg.

Figure 9: [15-to-15] FCT results under the data mining workload.



(a) Overall avg. (b) Small avg.
(c) Small tail (d) Large avg.

Figure 10: [14-to-1] FCTs under web search workload.



(a) Overall avg. (b) Small avg.
(c) Small tail (d) Large avg.

Figure 11: [14-to-1] FCTs under data mining workload.

is because they lack efficient scheme to gracefully utilize the available bandwidth in the network and do not take advantage of in-network priorities for scheduling. We further find that PPT performs better than Homa-Linux in small flow performance for both workloads. Compared to Homa-Linux, PPT reduces small flows' average/tail FCT by up to 84.5%/96.8% for the web search workload and 74.3%/95.1% for the data mining workload.

Remarks: *Readers may question that PPT performs better than Homa-Linux in the data mining workload, which contradicts the claimant in §4.3. This is mainly because of the inefficient implementation of the Homa-Linux network stack. First, it uses the GRO function to group multiple messages/flows into a batch. This will significantly increase the latency of small flows under the data mining workload with polarized flow sizes. Second, Homa-Linux implementation can only detect packet loss using timeout events and does not support advanced mechanisms like duplicated ACKs and SACK. These indirectly demonstrate that refactoring the TCP/IP network stack to implement a complete new one in the Linux kernel is not an easy task.*

Large flow's performance: PPT's performance improvements in small flows will not penalize (100KB, ∞) large flows. Specifically, we observe that PPT delivers the best performance for large flows, with the reduction in the average FCT of large flows over Homa-Linux, RC3, and DCTCP being up to 80.2%, 80.3%, 98.1%, respectively. These results indirectly demonstrate that PPT can well utilize the available bandwidth in the network and its scheduling design will not let large flows get starved.

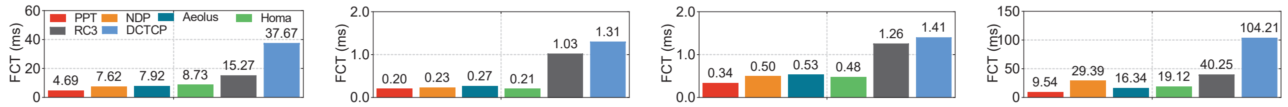
6.1.2 *14-to-1 Traffic Pattern.*

We further use the 15 hosts in our testbed to construct a 14-to-1 incast traffic pattern. In particular, we run the client application on one host to periodically send requests to the other fourteen hosts that run server applications and respond with requested data following the distributions of web search and data mining workloads. We choose a moderate 0.5 network load for this experiment. The results are shown in Fig.10 and Fig.11.

Overall FCT performance: Under the 14-to-1 traffic pattern, PPT still delivers the lowest overall average FCT among all schemes. Compared to Homa-Linux, RC3 and DCTCP, it reduces the average FCT of all flows by 74.8%, 92.7% and 95.5% for the web search workload and 32%, 23.4% and 94% for the data mining workload. The reason why PPT shows lower overall FCTs in incast scenarios is as follows. First, PPT's intermittent loop initialization design could accurately match the spare bandwidth's emergence. As a result, PPT will not send a burst of opportunistic packets and cause severe congestion as Homa-Linux and RC3 did. Second, even if congestion arises in LCP, PPT can quickly recover and fall back to DCTCP by suspending opportunistic packet transmission with EWD and ECN.

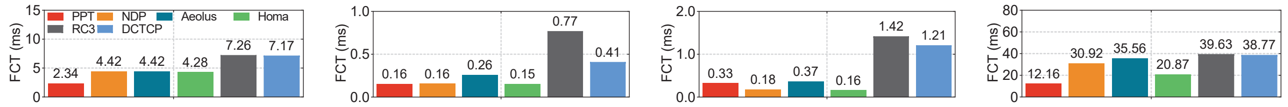
Small flow's performance: PPT also achieves lower FCTs for the small flows. Compared to RC3 and DCTCP, it reduces the average/tail FCT of small flows by 99.2%/99.9% and 99.3%/99.2%, respectively, for the web search workload. For the data mining workload, its reduction in the average/tail FCT over these two schemes also reaches 94.7%/99.3% and 84.9%/99.3%. Note here, RC3 even performs worse than DCTCP in some cases. This is because RC3 sends too many low-priority opportunistic packets to utilize the spare bandwidth, which causes severe packet losses in such many-to-one incast scenarios and, in turn, harms the high-priority packet transmission. Compared to Homa-Linux, PPT reduces the average/tail FCT of small flows by 72.9%/88.7% and 80.2/98% under the web search and data mining workloads, respectively.

Large flow's performance: Under the 14-to-1 incast scenario, PPT will not penalize large flow's performance. Specifically, compared



(a) Overall avg. (b) Small avg. (c) Small tail (d) Large avg.

Figure 12: [Simulation] FCT results under web search workload.



(a) Overall avg. (b) Small avg. (c) Small tail (d) Large avg.

Figure 13: [Simulation] FCT results under data mining workload.

to Homa-Linux, RC3 and DCTCP, PPT reduces the average FCT of large flows by up to 75.9%, 82.2%, and 95.1%, respectively, across the two workloads.

6.2 Large-scale Simulations

Comparisons: In addition to DCTCP, RC3 and Homa, we also compare PPT with two recent proactive transports: NDP [15] and Aeolus [17] (integrated with Homa by default).

Settings: We simulate a 1.4:1 oversubscribed topology and use all-to-all traffic patterns. This topology consists of 144 servers, 9 leaf switches and 4 spine switches, with the host and core links operated at 40 and 100Gbps, respectively. Unless otherwise specified, we use 0.5 load. We configure a per-port buffer of 120KB at the switches. For DCTCP, we set its ECN marking threshold to 96KB. For RC3, we set the send buffer to its recommended value—2GB. For PPT, we configure its send buffer to 2GB to keep consistent with RC3, but it can work well under smaller send buffer sizes (see §6.3). We set the ECN marking thresholds for PPT's HCP and LCP loops to 96KB and 86KB, respectively. Homa's simulations use an infinite switch buffer and lack a loss recovery mechanism. So, we use Aeolus's simulator with a timeout-based loss recovery mechanism to evaluate Homa. We set RTTbytes to 45KB for Homa and Aeolus to match our 40/100G topology, and the degree of overcommitment is 2. For NDP, we use the simulator provided by the authors with their recommended configurations. We also use the web search [34] and data mining [13] workloads.

Fig. 12 and Fig. 13 show the overall average FCT, average/tail FCT of ($\leq$ 100KB) small flows, and the average FCT of ($>$ 100KB) large flows for the web search and data mining workloads, respectively.

Overall FCT performance: PPT achieves the lowest average FCT of all flows. Compared to NDP, Aeolus, Homa, RC3, and DCTCP, PPT reduces the overall average FCT by 38.5%, 40.8%, 46.3%, 69.3%, and 87.5%, respectively, for the web search workload and 47.1%, 47.1%, 45.3%, 67.8%, and 67.4%, respectively for the data mining workload. The improvement of PPT over DCTCP is due to its use of LCP loop to efficiently utilize the available bandwidth. The reasons why PPT performs better than other schemes are that PPT can gracefully utilize the spare bandwidth without causing bandwidth waste like NDP or sending opportunistic packets too aggressively, like Aeolus, Homa, and RC3.

Small flow performance: For (0, 100KB] small flows, we find that PPT significantly outperforms RC3 and DCTCP, with the improvement in the average/tail of small flows by up to 80.1%/76.8% and 84.7%/75.9%, respectively, across the workloads. Note that RC3 even performs worse than DCTCP under the data mining workload. This is because it sends too excess low-priority packets in each RTT, crippling its desirable property in utilizing the spare bandwidth left by DCTCP. We further observe that compared to NDP, Homa, and Aeolus, PPT delivers comparable average FCT of small flows with them but maintains 32%, 35.8%, and 29.2% lower tail FCTs, respectively, for the web search workload. For the data mining workload, PPT is better than Aeolus but worse than Homa by 6.3%/54.5% in the average/tail FCT of small flows and worse than NDP by 45.5% in the tail FCT of small flows. This is because the data mining workload has polarized flow sizes, making Homa's aggressive and NDP's passive transmissions in the 1st RTT less likely to cause bursty. Whereas, Aeolus sends unscheduled packets at line rate but drops them immediately once bandwidth is used up, thus leading to small flows still exhibiting degraded performance.

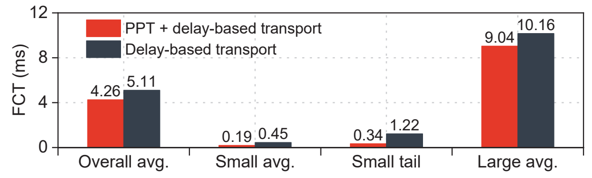


Figure 14: [Simulation] PPT works well with a delay-based transport conceptually equivalent to Swift [21].

Large flow performance: For large flows, we observe that PPT achieves the best performance, with the reduction in the average FCT of large flows over NDP, Aeolus, Homa, RC3 and DCTCP being up to 67.5%, 65.8%, 50.1%, 76.3%, and 90.8%, respectively. These results verify that PPT can achieve its design goal and bring no penalties on large flows.

Working with delay-based transport: PPT is designed around DCTCP. We now investigate if PPT's design can be used as a building block for other reactive transports. To this end, we use the ns-3 simulator to implement a variant of PPT on top of a delay-based transport (conceptually equivalent[7] to Swift [21]). Specifically, this variant starts an LCP loop whenever a flow's transmission delay falls below the target delay and closes it when it does not receive ACKs for two consecutive RTTs. Moreover, this variant uses the same flow scheduling method as PPT. Fig 14 compares the FCT statistics of this variant with those of the original delay-based transport

[7]Since the ns-3 simulator cannot accurately simulate the host congestion, it uses the same window adjustment algorithm as Swift [21] but adjusts the congestion window only based on the fabric delay.

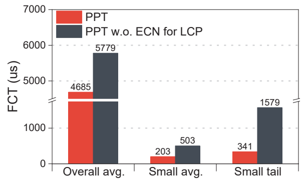


Figure 15: [Simulation] PPT w.o. LCP ECN

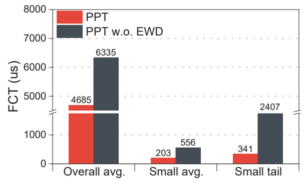


Figure 16: [Simulation] PPT w.o. EWD

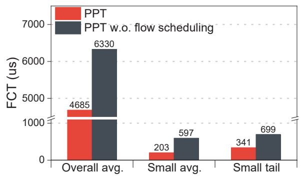


Figure 17: [Simulation] PPT w.o. scheduling

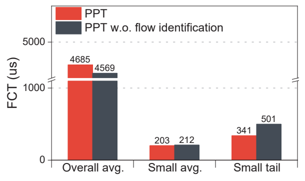


Figure 18: [Simulation] PPT w.o. identification

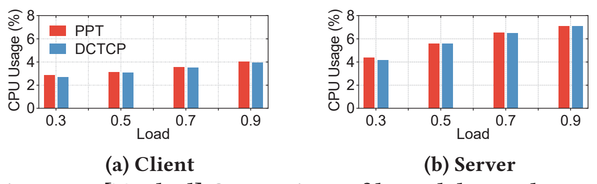


Figure 19: [Testbed] Comparison of kernel datapath processing overhead under web search workload.

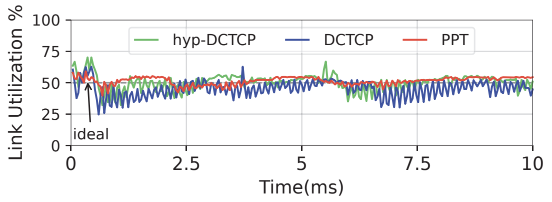


Figure 20: Link utilization under web search workload [34] at 0.5 load (ideal utilization is 50%).

under web search workload at 0.5 load. We find that when incorporating PPT's design with the original delay-based transport, the overall average FCT, the average/tail FCT of small flows, and the average FCT of large flows can be reduced by 16.7%, 56.5%/72.1%, and 11%, respectively. These results show that PPT works well with delay-based transport. We have also discussed PPT's compatibility with other transports. Please see appendix B.

Comparisons with other schemes: We also compare PPT with other schemes and the results showe that PPT outperforms RC3 even when limiting the available buffer for low-priority queues of RC3 and also performs better than PIAS [9] and HPCC [25] (see appendix D for more details).

6.3 PPT Deep Dive

6.3.1 Effectiveness of PPT's Design Components.

Effect of ECN for LCP loop: PPT enables ECN to control LCP loop to not affect HCP. To show its effect, we conduct an ns-3 simulation with the web search workload at 0.5 load using the same 40/100G topology as in 6.2. Figure. 15 shows the results. As we can see, without ECN, PPT's LCP loop may perceive congestion after packet loss, thus aggressively injecting LCP packets and harming HCP packet transmissions. As a result, the overall average FCT and the average/tail FCT of small flows achieved by PPT can be slowed down by 18.9%, 59.6%/78.4%, without ECN for the LCP loop.

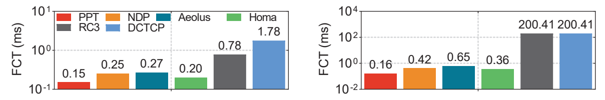
Effect of EWD: We further investigate the effectiveness of PPT's EWD design. We construct a PPT variant that does not use EWD and instead sends opportunistic packets at line rate once the LCP loop is opened. We use the same topology and workload as above. The results are shown in Figure 16. We find that PPT's performance significantly decreases when we disable EWD for it, with the overall average and the average/tail of small flows prolonged by 26% and 63.5%/85.8%, respectively. This demonstrates that the EWD design can contribute effectively to PPT's performance.

Effect of flow scheduling: PPT uses buffer-aware flow scheduling to assign higher priorities for flows with fewer bytes. To show its effect, we construct another variant of PPT that assigns packets with the same priorities. Similarly, we use the same topology and workload as above and show the results in Fig. 17. We find that compared to this variant, the original PPT can reduce the overall average FCT and average/tail FCT of the small flows by 26% and 66%/51.2%, respectively. This verifies the effectiveness of PPT's flow scheduling component.

Effect of flow identification: PPT's flow scheduling relies on a buffer-aware identification approach to identify large flows. To evaluate if this can benefit PPT's performance, we construct a PPT variant that turns off this approach and considers all flows non-identified. Fig. 18 compares the FCT statistics achieved by this variant and the original PPT for the web search workload under load 0.5. We have two findings. First, this variant delivers a slighter lower overall average FCT than PPT. This is because large flows in this variant have higher priorities than the original PPT. Second, the original PPT outperforms this variant in small flow performance, with the average/tail FCT of small flows reduced by 4.3%/31.9%. This is expected because, under this variant, both large and small flows will be initially mapped to the highest priority queue and gradually demoted to lower-priority ones, thus making large flows compete for bandwidth with small ones.

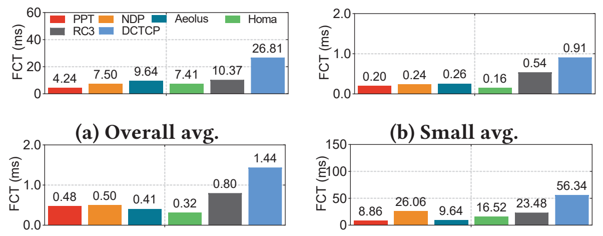
Overhead: Readers may question whether PPT incurs high CPU overhead. To answer this question, we measure the kernel space CPU overhead of PPT and DCTCP in the our testbed under the web search workload. Fig. 19 shows the client-side and server-side CPU overhead for PPT and DCTCP. In all cases, we observe that PPT only incurs a slightly higher CPU usage than DCTCP. The largest gap between their CPU usage is less than 1%. On a closer analysis, we find that the average gap between the CPU usage of PPT and DCTCP decreases on both client and server as the load increases. This is because, compared to DCTCP, the increased overhead from PPT is caused by the low-priority opportunistic packet transmission, while a larger load leads to less spare bandwidth and, thus lower overhead gap.

Effect of spare bandwidth utilizing: PPT aims to utilize the spare bandwidth of DCTCP. We run an ns-3 simulation with the same setting as that for Fig. 1 to quantify this point. We compare the link utilization of PPT with DCTCP and the hypothetical DCTCP (as described in §2.3). Again, the load is set to 0.5, and an ideal link utilization should be 50%. Fig. 20 shows the link utilization achieved by different schemes. We find that PPT achieves almost the same link utilization as the hypothetical DCTCP; that said, both



(a) Small avg. (b) Small tail

Figure 21: [Simulation] FCT results with an Memcached workload (where all flows are less than 100KB).



(a) Overall avg. (b) Small avg.

(c) Small tail (d) Large avg.

Figure 22: [100/400G topology] FCT results under web search workload.

maintain the utilization around 50%. However, DCTCP can drop to 25% utilization, resulting in up to 1.8× lower utilization than our PPT. Moreover, the average utilization in the steady state achieved by PPT is 15% higher than that of DCTCP.

6.3.2 *Handling Some Extreme Cases.*

Working with tiny workload: We further evaluate PPT under an Memcached workload from Facebook, which is exactly the W1 workload used in Homa [32]. In this workload, more than 70% of flows are less than 1000bytes, and all flows are less than 100KB. Fig. 21 depicts the FCT statistics of different schemes under the same topology as above with this memcached workload at 0.5 load. We find that PPT achieves the best performance, with the average/tail FCT reduced by at least 25%/55.6%, compared to all the other tested schemes. PPT can utilize available bandwidth gracefully and schedule flows with in-network priorities, thus performing better than DCTCP and RC3. The reason why the remaining three proactive transports perform worse than PPT stems from the fact that this Memcached workload is composed of small flows. So, not sending data in the first RTT like NDP will cause bandwidth waste, while sending data at line rate like Homa and Aeolus will lead to bursty; both cases bring slowdown for small flows.

Working with 100/400G topology: We further evaluate PPT in higher line rates. We still use the two-tier topology as in §6.2 but the host and core links are operated at 100Gpbs and 400Gbps, respectively. Fig.22 shows the FCT results of different schemes under the web search workload at 0.5 load. We have the following findings. First, compared to NDP, Aeolus, Homa, RC3, and DCTCP, PPT reduces the overall average FCT by 43.5%, 56%, 42.8%, 59.1%, and 84.2%, respectively, and also reduces the average FCT of large flows by 66%, 8.1%, 46.4%, 62.3%, and 84.3%, respectively. Second, PPT achieves 16.7%, 23.1%, 63%, and 78% lower average FCT of small flows than NDP, Aeolus, RC3, and DCTCP, respectively. Third, PPT delivers higher tail FCTs of small flows than Homa and Aeolus. The key reason is that a higher line rate leads to a higher value of BDP. This means that small flows could inject more opportunistic packets into the network via the LCP loop, thus making small flows in PPT being more likely to suffer from higher tail FCT.

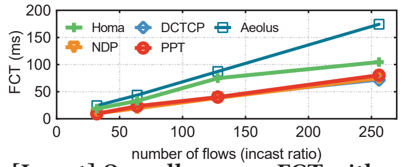


Figure 23: [Incast] Overall average FCT with varying incast ratio (RC3 is not included as it cannot sustain heavy incast).

Working with highly bursty workloads: We study the behavior of PPT under highly bursty scenario, where we construct N-to-1 incast traffic pattern in the above oversubscribed topology (N= 32, 64, 128 and 256). We compare PPT with NDP, Homa, Aeolus and DCTCP. Note here RC3 is not included as it cannot sustain heavy incast. All flows are generated according to the Web Search workload at 0.6 network load. We choose N senders randomly across all servers, and one server as the receiver. Fig. 23 shows the average FCT of all flows. We make two observations. First, PPT achieves similar performance with DCTCP. This is expected because PPT aims to utilize the spare bandwidth in the first few RTTs and queue buildup phase. However, in heavy incast scenario, the maximum window size would be very small for each flow, thus leaving little spare bandwidth in the first few RTTs. Further, the high priority traffic almost saturate the switch buffer under heavy incast, leaving little space for low priority opportunistic packets. As a result, PTT will fall back to DCTCP and can hardly make further improvement. Second, PPT can achieve better performance than Homa and Aeolus while similar performance with NDP. This is because PPT uses ECN to quickly stop low priority packet transmission, while NDP cuts payload and reserves packet headers, to maintain low queuing delay for normal packet transmission. In contrast, Homa blindly transmits a BDP amount of data for each new flow in the 1st RTT, which would cause sporadic traffic spikes, non-trivial queuing delay and eventually packet losses under heavy incast. Aeolus de-prioritizes first-RTT unscheduled packets and sends them at line-rate, which could be dropped once bandwidth is used up. Though these lost unscheduled packets will not trigger retransmissions, they lead to bandwidth waste and degrade the overall FCTs as well.

Working with non-oversubscribed topology: PPT also behaves well under non-oversubscribed topology (see appendix E).

6.3.3 *Sensitivity Analysis.*

PPT works well under different send buffer sizes and ECN marking thresholds. Due to the page limit, we move the results to appendix F.

7 CONCLUSION

PPT is a pragmatic DCN transport that uses a dual-loop rate control design to gracefully utilize the available bandwidth and further complements its design with a buffer-aware flow scheduling to optimize small flow's performance. We have implemented a PPT prototype using commodity hardware, and evaluated it through small-scale testbed experiments and large-scale simulations. Extensive evaluations show that PPT is a viable solution that achieves our design goals. *This work does not raise any ethical issues.*

Acknowledgements: This work is supported by NSFC under Grant 62202325. We thank our shepherd Praveen Kumar and the anonymous SIGCOMM reviewers for their constructive feedback and suggestions.

REFERENCES

[1] ACID: Distributed Service Governance Framework. https://github.com/zavier-wong/acid.

[2] DCTCP in Linux Kernel 3.18, 2014. https://kernelnewbies.org/Linux_3.18.

[3] DCTCP in Windows Server, 2012. http://technet.microsoft.com/en-us/library/hh997028.aspx.

[4] Memcached. http://memcached.org/.

[5] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data center tcp (dctcp). In *SIGCOMM* (2010).

[6] Alizadeh, M., Kabbani, A., Edsall, T., Prabhakar, B., Vahdat, A., and Yasuda, M. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI* (2012).

[7] Alizadeh, M., Yang, S., Sharif, M., Katti, S., McKeown, N., Prabhakar, B., and Shenker, S. pfabric: Minimal near-optimal datacenter transport. In *SIGCOMM* (2013).

[8] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload analysis of a large-scale key-value store. In *Proceedings of ACM SIGMETRICS* (2012).

[9] Bai, W., Chen, L., Chen, K., Han, D., Tian, C., and Wang, H. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *NSDI* (2015).

[10] Bai, W., Hu, S., Chen, K., Tan, K., and Xiong, Y. One more config is enough: Saving (DC) TCP for high-speed extremely shallow-buffered datacenters. In *INFOCOM* (2020).

[11] Cho, I., Jang, K., and Han, D. Credit-scheduled delay-bounded congestion control for datacenters. In *SIGCOMM* (2017).

[12] Dukkipati, N., Refice, T., Cheng, Y., Chu, J., Herbert, T., Agarwal, A., Jain, A., and Sutin, N. An argument for increasing TCP's initial congestion window. *ACM SIGCOMM Computer Communication Review 40*, 3 (2010), 26–33.

[13] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. VL2: A scalable and flexible data center network. In *SIGCOMM* (2009).

[14] Grossman, L. Large receive offload implementation in neterion 10gbe ethernet driver. In *Linux Symposium* (2005), p. 195.

[15] Handley, M., Raiciu, C., Agache, A., Voinescu, A., Moore, A. W., Antichi, G., and Wójcik, M. Re-architecting datacenter networks and stacks for low latency and high performance. In *SIGCOMM* (2017).

[16] Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., and Miller, D. The express data path: Fast programmable packet processing in the operating system kernel. In *CoNEXT* (2018).

[17] Hu, S., Bai, W., Zeng, G., Wang, Z., Qiao, B., Chen, K., Tan, K., and Wang, Y. Aeolus: A Building Block for Proactive Transport in Datacenters. In *SIGCOMM* (2020).

[18] Jorgensen, S., Holodnak, J., Dempsey, J., de Souza, K., Raghunath, A., Rivet, V., DeMoes, N., Alejos, A., and Wollaber, A. Extensible machine learning for encrypted network traffic application labeling via uncertainty quantification. *IEEE Transactions on Artificial Intelligence* (2023).

[19] Judd, G. Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter. In *NSDI* (2015).

[20] Kaufmann, A., Stamler, T., Peter, S., Sharma, N. K., Krishnamurthy, A., and Anderson, T. TAS: TCP acceleration as an OS service. In *EuroSys* (2019).

[21] Kumar, G., Dukkipati, N., Jang, K., Wassel, H. M., Wu, X., Montazeri, B., Wang, Y., Springborn, K., Alfeld, C., Ryan, M., et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of ACM SIGCOMM* (2020).

[22] Lee, C., Park, C., Jang, K., Moon, S., and Han, D. Accurate latency-based congestion feedback for datacenters. In *ATC* (2015).

[23] Li, Q., Dong, M., and Godfrey, P. B. Halfback: Running short flows quickly and safely. In *CoNEXT* (2015).

[24] Li, W., Xie, X., Liu, Y., Li, K., Chen, K., Ge, Z., Qi, H., Zhang, S., and Liu, G. Flow scheduling with imprecise knowledge. In *Proceedings of USENIX NSDI* (2024).

[25] Li, Y., Miao, R., Liu, H. H., Zhuang, Y., Feng, F., Tang, L., Cao, Z., Zhang, M., Kelly, F., Alizadeh, M., et al. HPCC: High precision congestion control. In *SIGCOMM* (2019).

[26] Liu, D., Allman, M., Jin, S., and Wang, L. Congestion control without a startup phase. In *Proc. PFLDnet* (2007).

[27] McCanne, S., and Jacobson, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter* (1993), vol. 46.

[28] Meng, T., Schiff, N. R., Godfrey, P. B., and Schapira, M. Pcc proteus: Scavenger transport and beyond. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 615–631.

[29] Mittal, R., Lam, V. T., Dukkipati, N., Blem, E., Wassel, H., Ghobadi, M., Vahdat, A., Wang, Y., Wetherall, D., and Zats, D. TIMELY: RTT-based congestion control for the datacenter. In *SIGCOMM* (2015).

[30] Mittal, R., Sherry, J., Ratnasamy, S., and Shenker, S. Recursively cautious congestion control. In *NSDI* (2014).

[31] Mohammad, A., Adel, J., and Balaji, P. Analysis of DCTCP. *ACM SIGMETRICS Performance Evaluation Review* (2011).

[32] Montazeri, B., Li, Y., Alizadeh, M., and Ousterhout, J. Homa: A receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM* (2018).

[33] Ousterhout, J. A linux kernel implementation of the homa transport protocol. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (2021), pp. 99–115.

[34] Roy, A., Zeng, H., Bagga, J., Porter, G., and Snoeren, A. C. Inside the social network's (datacenter) network. In *SIGCOMM* (2015).

[35] Singh, A., Ong, J., Agarwal, A., Anderson, G., Armistead, A., Bannon, R., Boving, S., Desai, G., Felderman, B., Germano, P., et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *SIGCOMM* (2015).

[36] Vamanan, B., Hasan, J., and Vijaykumar, T. Deadline-aware datacenter tcp (d2tcp). In *SIGCOMM* (2012).

[37] Vanini, E., Pan, R., Alizadeh, M., Taheri, P., and Edsall, T. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *NSDI* (2017).

[38] Wang, Z., Luo, L., Ning, Q., Zeng, C., Li, W., Wan, X., Xie, P., Feng, T., Cheng, K., Geng, X., et al. {SRNIC}: A scalable architecture for {RDMA}{NICs}. In *Proceedings of USENIX NSDI* (2023).

[39] Wu, H., Feng, Z., Guo, C., and Zhang, Y. Ictcp: Incast congestion control for tcp in data-center networks. *IEEE/ACM transactions on networking 21*, 2 (2012), 345–358.

[40] Wu, H., Ju, J., Lu, G., Guo, C., Xiong, Y., and Zhang, Y. Tuning ECN for data center networks. In *CoNEXT* (2012).

[41] Yasukata, K., Honda, M., Santry, D., and Eggert, L. StackMap:Low-Latency Networking with the OS Stack and Dedicated NICs. In *ATC* (2016).

[42] Zhao, Y., Saeed, A., Zegura, E., and Ammar, M. Zd: A scalable zero-drop network stack at end hosts. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies* (New York, NY, USA, 2019), CoNEXT '19, Association for Computing Machinery, p. 220–232.

[43] Zhu, Y., Eran, H., Firestone, D., Guo, C., Lipshteyn, M., Liron, Y., Padhye, J., Raindel, S., Yahia, M. H., and Zhang, M. Congestion control for large-scale RDMA deployments. In *SIGCOMM* (2015).

Parameter	Setting
Switch buffer size	50MB
Switch port number	54
RTT	80μs
RTO_{min}	10ms
RTTbytes for Homa	50KB
Overcommitment degree for Homa	2
DCTCP's ECN threshold	Default: 100KB
HCP's ECN threshold	Default: 100KB
LCP's ECN threshold	Default: 80KB
Identification threshold	Default: 100KB

Table 3: Testbed parameters.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A PARAMETER SETTING IN TESTBED

Tabe 3 summarizes the parameter settings in our testbed experiments.

B DISCUSSION

Compatible with other transports: PPT is based on ECN-style transport—DCTCP, and we have also shown that PPT can be integrated into delay-based transport (appendix E). Actually, PPT's design may also be used as a building block for INT-based transport like HPCC [25]. For example, one may open a PPT's LCP loop to send low-priority opportunistic packets whenever HPCC's estimated in-flight bytes are smaller than BDP and use PPT's buffer-aware scheduling to prioritize small flows over large ones for lower FCTs. For proactive transport like Homa [32], no available signal can be used to identify the congestion status in the network core. So, one may have no idea when to start PPT's LCP loop, and accordingly, integrating PPT's design with proactive transport remains an interesting open problem.

Portability issue: PPT is implemented in the Linux Kernel and is easy to deploy. Albeit this, we envision that it suffers from portability issues, which means that porting PPT to a different OS Kernel vision will require additional engineering effort. One possible way to tackle this problem is to implement PPT in a kernel-bypass manner, i.e., restoring eBPF [27] and XDP [16]. In practice, the only technical challenge is to guarantee the feasibility of controlling opportunistic packets, but this could involve the cooperation between kernel-bypass manners and the network stack.

C OTHER RELATED WORK

We have discussed the closely related work extensively in this paper. Here, we only provide some more points about the proactive transports and some other ideas that have not been discussed elsewhere.

For proactive transport like Homa [32], there are kernel-based implementations, such as Homa/Linux [33]. However, this needs to implement many network subsystems such as GRO/GSO and state/memory management, incurring ~10k lines of code (see Table 4). Even though this can be done, some network systems (e.g., TSO [42] and LRO [14]) are not compatible. Worse still, one must significantly change the application's code to use Homa/Linux [33]. We dissect the code organization of a distributed key-value store

Module	Lines of Code	Percentages
User API	1900	15%
Transport control	2800	22%
GRO/GSO	400	3.1%
State management	700	5.5%
Memory management	300	2.4%
Timeout retransmission	300	2.4%
Other	6300	49.6%

Table 4: Lines of code for Homa/Linux stack.

Modules	Lines of Code	Modified?
Socket	2080	Y
HTTP package header processing	1516	N
RPC	975	Y
RAFT consensus protocol	1365	N
Coroutine synchronization	145	N
IO	393	Y
Other	1694	N

Table 5: Lines of code to be changed for a key-value store application to run on Homa/Linux stack.

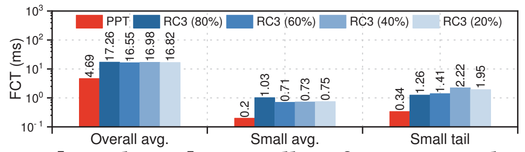


Figure 24: [Simulation] RC3 still performs worse than PPT even when limiting the available buffer for its low-priority queues.

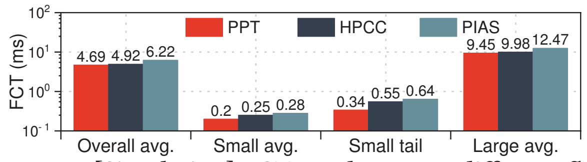


Figure 25: [Simulation] FCT results across different flow types achieved by HPCC, PIAS and PPT.

application built with the RAFT consensus protocol [1]. Table 5 lists the modules in this application and their lines of code required to be modified for using Homa/Linux [33]. We find that three modules (i.e., socket, RPC, and IO) must be modified, which requires 3448 lines of code, accounting for 42.2% of the entire application code.

There are other efforts that improve the performance of DCN, such as load balancing (e.g., CONGA [7], Flowlet [37]), and TCP/IP acceleration (e.g., StackMap [41], TAS [20]). These designs may work orthogonally with PPT to further improve the performance. There are some other reactive transports (e.g., HULL [6], D^2TCP [36], DX [22], DCQCN [43], TIMELY [29], Swift [21]) that have not been discussed. These transports generally require multiple rounds to converge to the correct rate, and lack efficient flow scheduling scheme to optimize small flow's performance.

D COMPARISONS WITH OTHER SCHEMES

Comparisons with RC3 variant: Readers may question if RC3 would still perform poorly if we limit the switch buffer that its low-priority queues could use. To verify this point, we vary the buffer for RC3's low priority traffic from 20% to 80% of the switch buffer, and run a simulation to compare its FCT results with those of PPT in Fig. 24. We find that PPT significantly outperforms RC3 regardless

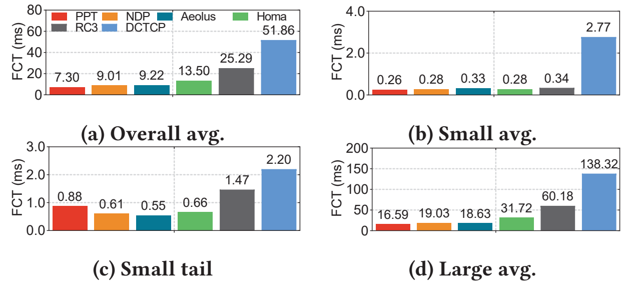


Figure 26: [Non-oversubscribed topology] FCT results under the web search workload.

of the available buffer for RC3's low-priority traffic. Compared to RC3 across all shown cases, PPT reduces the overall average FCT and the average/tail FCT of small flows by up to 71% and 73%/75%, respectively. The reason is that RC3 makes no attempt to protect high-priority traffic and lets the low-priority loop keep sending opportunistic packets until it crosses with the primary control loop. So, when limiting the available buffer for RC3's low-priority queues, the vast majority of opportunistic packets will be dropped, and RC3's low-priority loop may quickly close, thus showing little effect on utilizing spare bandwidth.

Comparisons with PIAS and HPCC: We now compare our PPT with HPCC [25] and PIAS [9]. Fig. 25 shows the FCT statistics across different flow types for HPCC, PIAS and PPT. We find that PPT outperforms PIAS in the FCTs across all flow sizes. Specifically, it reduces the overall average FCT, the average/tail FCT of small flows, and the average FCT of large flows by 24.6%, 28.6%/46.9%, and 24.2%, respectively. The reasons for these results are two-fold. First, PIAS's rate control is based on DCTCP, which cannot efficiently utilize the available bandwidth. Second, PIAS's scheduling can only prioritize small flows over large ones in later transmission stages, i.e., after the large flows transmit a significant amount of data. As the second observation, PPT also performs slightly better than HPCC, with the overall average FCT, the average/tail FCT of small flows, and the average FCT of large flows reduced by 4.7%, 20%/38.2%, and 5.3% respectively. This outcome is mainly because HPCC does not take advantage of the in-network priorities to schedule flows.

E WORKING WITH NON-OVERSUBSCRIBED TOPOLOGY

Existing proactive transports like Homa [32] and Aeolus [17] are not designed for oversubscribed network, because they assume the network is fully provisioned with no congestion in the core network. To assess if PPT still preserves performance gains in an environment that is more friendly to existing proactive transports, we simulate a non-oversubscribed topology. In this topology, there are 9 leaf switches and 4 spine switches. Each leaf switch is connected to 16 hosts with 10Gbps links and 4 spine switches with 40Gbps links. We use the web search workload for this experiment. Fig. 26 depicts the results. From this figure, we observe that PPT achieves the best performance for both overall flows and large flows, with the overall average FCT reduced by 19%, 20.8%, 45.9%, 71.1%, and 85.9% and the average FCT of large flows reduced by 12.8%, 11%, 47.7%, 72.4%, and 88%, compared to NDP, Aeolus, Homa, RC3, and DCTCP, respectively.

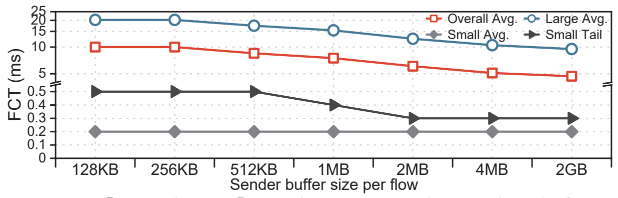


Figure 27: [Simulation] PPT's FCT results under different TCP sender buffer capacity sizes.

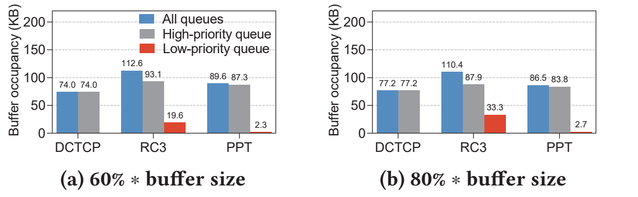


Figure 28: [Simulation] Buffer occupancies under different ECN marking thresholds.

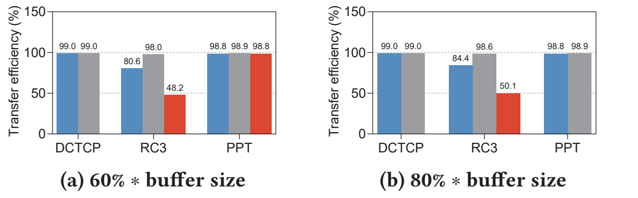


Figure 29: [Simulation] Transfer efficiencies under different ECN marking thresholds.

We further find that PPT still delivers a slightly lower average FCT of small flows than NDP, Aeolus, and Homa. However, the tail FCT of small flows achieved by PPT is at most 37.5% worse than the three proactive transports. The reasons for the results above are two-fold. First, PPT delays the LCP loop initialization to the second RTT for identified large flows, thus making small flows have relatively more bandwidth in the 1st RTT and accordingly maintaining comparable average FCT of small flows. Second, under non-oversubscribed topology, congestion occurs primarily at the last-hop, and proactive transport like Homa can use credit and precise flow size information to emulate SRPT efficiently. By contrast, PPT separates HCP and LCP into different priority queues, which could prioritize HCP packets of identified large flows over LCP packets of small flows. Worse still, PPT leaves some unidentified large flows, which could coexist with small flows in higher priority queues until they are gradually moved to lower priority queues, thus degrading the tail FCT of small flows.

F SENSITIVITY ANALYSIS

Impact of send buffer: The accuracy of PPT's large flow identification relies on having large send buffers. One may question the impact of send buffer capacity on PPT's performance. To validate this, we configure multiple buffer sizes ranging from 128KB to 2G and show the FCT results of PPT in Fig. 27 under the web search workload at 0.5 load and the oversubscribed topology. When the buffer size is only 128KB, the average/tail FCT of small flows achieved by PPT is 0.20/0.48ms, which still lower than the proactive transports (see NDP, Aeolus and Homa results in Fig. 13b and

Fig. 13c). On the other hand, the overall average FCT and the average FCT of large flows are 10ms and 20.2ms respectively, which are slightly higher than those of proactive transports. Once the buffer size reaches 2MB, both the overall average FCT and the average FCT of large flows of PPT can fall below those of NDP, Aeolus and Homa. Note that 2MB is very small as compared to the host memory, but is large enough for holding most flows as the average flow size of the web search workload is 1.6MB. Further increasing the send buffer to 4MB does provide significant performance improvement.

Impact of ECN marking threshold: We first run ns-3 simulations with two senders and one receiver to investigate the impact of ECN marking threshold on switch buffer occupancies. The bottleneck link capacity is 40Gbps, and the total switch buffer is set to 120KB. Fig. 28 shows the buffer the high-priority queue occupies compared to the buffer the low-priority queue occupies under different ECN marking thresholds (i.e., 60% and 80% of the total buffer size). Note here this experiment configures the same ECN marking threshold for both low-&high-priority control loops. We make the following findings from this figure. First, PPT requires 20.4%∼21.6% less switch buffer than RC3. Moreover, the low-priority queue under PPT only occupies 2.6%∼3.1% of the total buffer occupancy, while RC3's low-priority queue accounts for 17.4%∼30.2%. Second, a larger ECN marking threshold makes RC3's low-priority queue occupy more buffer, while PPT could make its low-priority queue maintain relatively low and stable buffer occupancy under different ECN marking thresholds. Third, PPT incurs 10.8%∼17.4% more buffer than DCTCP, while it delivers lower FCT (see §6.1 and §6.2).

We further conduct an experiment to inspect the impact of ECN marking threshold on PPT's transfer efficiency. The settings are the same as the experiment of investigating the impact of ECN on buffer occupancy. We calculate transfer efficiency as the total received data bytes over the total sent bytes. The higher the transfer efficiency, the fewer packet losses. Fig. 29 shows the results. This figure shows that PPT achieves comparable transfer efficiency with DCTCP. Compared to RC3, PPT delivers14.6%∼18.4% higher transfer efficiency. On a closer analysis, we find that though the RC3 maintains only a little bit lower transfer efficiency, its low-priority transfer efficiency is 47.8%∼51.2% lower than PPT. This implies that in RC3, many low-priority opportunistic packets are dropped and require the normal DCTCP loop to retransmit. In this case, though the high-priority control loop's efficiency may not be affected, a considerable part of its efficiency is used for filling the hole left by the low-priority control loop, thus being ineffective in filling the spare bandwidth left by DCTCP.