TAURUS: Towards A High-Performance and Generic Congestion Control Framework for Datacenter Networks

LUYANG LI, Institute of Computing Technology, Chinese Academy of Sciences, China
HENG PAN, Computer Network Information Center, Chinese Academy of Sciences, China
PENGYI ZHANG and KAI LV, Institute of Computing Technology, Chinese Academy of Sciences, China
ZILONG WANG and XINCHEN WAN, Hong Kong University of Science and Technology, China
DAI ZHANG, XIAOLONG ZHONG, and HAORAN WEI, Douyin Co., Ltd., China
LICHAO LIU and HUICHEN DAI, Douyin Co., Ltd., China
QINGSONG NING, Researcher, China
XIN WEI and SHIDENG ZHANG, Douyin Co., Ltd., China
HONGTAO GUAN, Institute of Computing Technology, Chinese Academy of Sciences, China
ZHENYU LI, Institute of Computing Technology, Chinese Academy of Sciences, China and University of Chinese Academy of Sciences, China
GAOGANG XIE, Computer Network Information Center, Chinese Academy of Sciences, China and University of Chinese Academy of Sciences, China

Congestion control (CC) is crucial for datacenter networks (DCNs), and CC frameworks are proposed to enable users to easily deploy new algorithms tailored to diverse scenarios. The framework is desired to be *high-performance* and *generic*: (i) allows CC to achieve high throughput and low latency. (ii) supports various algorithms and congestion scenarios. However, prior works either suffer from *performance* limitations or lack sufficient *generality*. CCP experiences throughput degradation under heavy traffic, while DOCA-PCC improves performance using hardware but lacks support for detecting and mitigating host congestion.

In this paper, we present TAURUS, a high-performance and generic CC framework through the *hardware-software* co-design. To this end, TAURUS partitions CC functions into distinct tasks and maps them onto suitable hardware/software components while mitigating excessive interaction overhead. Specifically, TAURUS designs a collaborative signal collection mechanism to support diverse congestion feedback, a type-aware message report engine to reduce communication overhead, and software built-in handlers to facilitate deployments. We have implemented a fully functional TAURUS on commodity servers with FPGA-based NICs. Experimental results show that TAURUS supports various CC algorithms in achieving their near-native performance. Compared to CCP, TAURUS improves throughput by 32.3%, reduces latency by 96.4%, and lowers CPU overhead by 158.7%. Compared to DOCA-PCC, TAURUS improves throughput by 9.3% and reduces latency by 28.8%.

CCS Concepts: • **Networks → Data center networks**; **Network control algorithms**.

Additional Key Words and Phrases: Datacenter Network; Congestion Control;

ACM Reference Format:
Luyang Li, Heng Pan, Pengyi Zhang, Kai Lv, Zilong Wang, Xinchen Wan, Dai Zhang, Xiaolong Zhong, Haoran Wei, Lichao Liu, Huichen Dai, Qingsong Ning, Xin Wei, Shideng Zhang, Hongtao Guan, Zhenyu Li, Gaogang Xie. 2025. TAURUS: Towards A High-Performance and Generic Congestion Control Framework for Datacenter Networks. *Proc. ACM Netw.* 3, CoNEXT4, Article 26 (December 2025), 25 pages. https://doi.org/10.1145/3768973

.

ACM 2834-5509/2025/12-ART26
https://doi.org/10.1145/3768973

1 Introduction

Congestion control (CC) is the key in datacenter networks (DCNs) to ensure ultra-low latency and high throughput for applications, such as LLMs [30, 53] and graph analysis [13, 39]. Modern datacenters have proposed numerous CC algorithms [5, 9, 33, 36, 40, 78] tailored to different networks. Unfortunately, deploying new CC algorithms at a large scale is typically challenging due to their tight integration into the transport datapath (*e.g.*, RDMA [62], TCP [12], etc.). Modifications can be laborious, as they need to carefully consider datapath resources [69], security [7], performance constraints [52], and even require NIC architecture adjustment [9, 36] (see §2.1).

Therefore, there is a growing interest in both academia and industry in programmable CC frameworks [15, 52]. These frameworks decouple CC decisions from traditional datapaths, enabling users to easily deploy customized algorithms through a set of interfaces. Generally, users expect the framework to be **high-performance** and **generic**: (i) allows CC to achieve exceptional performance (*e.g.*, link-speed throughput and μs-level latency), comparable to in-datapath deployments; (ii) supports diverse CC algorithms (*a.k.a.* **algorithm-wise** generality) and handles various congestion scenarios (*a.k.a.* **congestion-wise** generality), including host congestion [2] and fabric congestion [33].

Prior works, such as CCP [52] and DOCA-PCC [15], have started to support multiple CC algorithms, but they either suffer from performance issues or lack sufficient generality. Specifically, CCP utilizes host CPUs to build separate agents outside of the software datapaths, which facilitate sophisticated CC algorithms [16, 72, 74]. However, escalating traffic (*e.g.*, up to 400Gbps) can exceed its processing capabilities, leading to performance degradation and increased overhead (§2.3). To address this issue, DOCA-PCC offloads congestion signal extraction and decision-making to programmable hardware (*i.e.*, BlueField [18]), which significantly speeds up the response time. Nonetheless, it falls short in dealing with host congestion, which widely impacts modern datacenter clusters [2, 3, 33, 35, 65]. Host congestion occurs between CPUs and peripherals due to host resource contention (*e.g.*, memory[35], PCIe bandwidth [2, 3], etc.). However, DOCA-PCC offloads signal collection to external hardware and mainly provides feedback from the fabric and NIC, which limits its ability to pinpoint the location and severity of host congestion. Moreover, DOCA-PCC merely adjusts the sending rate to the fabric and lacks support to regulate host resources to fundamentally resolve host congestion (§2.3).

Thus, we ask: *is it possible to design a high-performance and generic CC framework for DCNs?* To answer this question, we notice that achieving high performance favors *hardware* assistance to handle heavy traffic loads while ensuring generality requires *host software* involvement to accommodate different CC algorithms and congestion scenarios.

In this paper, we thus present Taurus, a **hardware-software co-design** CC framework to balance performance and generality requirements. To achieve this, we first propose a basic design principle that partitions CC functions into distinct tasks, and maps them onto suitable hardware/-software components, while mitigating excessive communication overhead (§3.1). Guided by this principle, we systematically analyze the conceptual model of CC in modern DCNs and categorize its tasks based on their performance and flexibility demands. The performance-critical tasks (*e.g.*, signal collection and delivery adjustment) are offloaded to the hardware to guarantee the datapath performance, whereas tasks requiring flexibility (e.g., decision-making and custom feedback) are handled in software (§3.2).

To enable *algorithm-wise* and *congestion-wise* generality, Taurus designs a collaborative signal collection mechanism (§3.3). Our *key insight* is to harness the hardware's ability to do best-effort signal extraction, while leaving fine-grained signal analysis to software. Specifically, the hardware extracts *predefined signals* widely used in existing CC algorithms, along with *feedback segments* that contain new user-defined signals; The software further parses custom signals from *feedback segments* and provides interfaces to collect *host signals*. To ensure high performance, TAURUS designs a type-aware message report engine with a two-tier aggregation mechanism to reduce the communication overhead between hardware and software (§3.4). Furthermore, TAURUS develops a built-in message handler to deal with exceptional reports, an algorithm handler to dispatch different feedback events to corresponding algorithms and filter out non-essential updates, thus facilitating the algorithm deployment (§3.5). We have implemented a fully functional TAURUS based on commodity servers with FPGA-based NICs and deployed it into DCNs at 400Gbps link speeds (§4). We evaluate TAURUS on both testbeds and large-scale simulations with the real-world workloads (§5). The results show that TAURUS supports a variety of CC algorithms and congestion scenarios, achieving near-native performance comparable to in-datapath CC deployments. Compared to CCP, TAURUS improves throughput by 32.3%, reduces latency by 96.4%, and lowers CPU overhead by 158.7%. Compared to DOCA-PCC, TAURUS improves throughput by 9.3% and reduces latency by 28.8%.

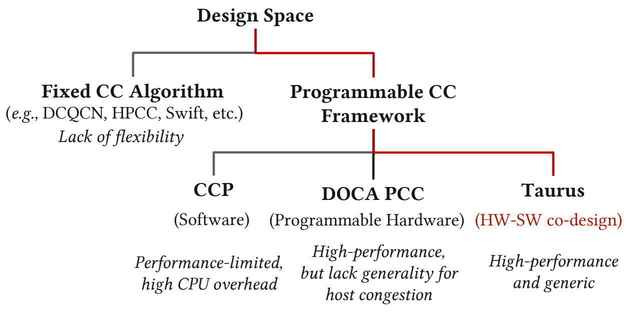

Design Space
Fixed CC Algorithm
(e.g., DCQCN, HPCC, Swift, etc.)
Lack of flexibility
Programmable CC Framework
CCP
(Software)
Performance-limited, high CPU overhead
DOCA PCC
(Programmable Hardware)
High-performance, but lack generality for host congestion
Taurus
(HW-SW co-design)
High-performance and generic

Fig. 1. Design space of congestion control for datacenter networks and the position of TAURUS.

Figure 1 summarizes the design space of CC for DCNs and highlights the position of TAURUS. Compared to the other solutions, TAURUS provides a generic and high-performance CC framework with hardware-software co-design. Our primary contributions are as follows:

- We experimentally analyze the limitations of previous works (§2.3) and present TAURUS, a hardware-software co-design architecture to achieve both generality and performance for CC frameworks (§3.1 & §3.2).
- We propose an innovative collaborative signal collection mechanism to support diverse CC feedbacks (§3.3), a type-aware message report engine with a two-tier aggregation to reduce communication overhead (§3.4), and software built-in handlers to simplify CC deployment (§3.5).
- We implement a fully functional TAURUS and deploy it into DCNs. Experiments with real-world workloads demonstrate its ability to achieve generality and high performance compared to other CC frameworks (§4 & §5).

2 Background and Motivation

2.1 The Needs for CC Frameworks

2.1.1 Keeping a Fixed CC Algorithm is Not Enough. Modern datacenter providers have proposed various CC algorithms [5, 9, 33, 36, 50, 78] in their DCNs. However, it is challenging to keep a fixed CC algorithm that consistently maintains high performance across all environments:

Distinct application scenarios. Different application scenarios exhibit distinct congestion sensitivities, traffic characteristics and workload patterns. For AI training on Dragonfly topologies [1], switches are organized into all-to-all groups, with intra-group flows traversing a single hop while inter-group paths may quickly become congested without adaptive routing [71]. In contrast, public services deployed on Clos networks [4] feature multiple equivalent paths with longer hop counts.

Therefore, CC should adjust its assumptions for distinct topologies. For traffic patterns, storage workloads generate more elephant flows [48], whereas RPC workloads [32] exhibit more mice flows. Thus, CC requires distinct strategies and parameters to suit varied scenarios.

(ii) Evolving Datacenter Networks. The datacenter network remains unremitting evolution. *First*, the increasing BDP allows more transfers to be completed quickly. As reported in [9], approximately 80% of RPCs can be completed in just one RTT. Thus, CC needs to increase regulation speed (*e.g.*, from RTT level to sub-RTT level). *Second*, network devices are iterating with advanced functionalities (*e.g.*, INT [24], packet trimming [56], etc.). Thus, it provides more opportunities for CC algorithms to improve their precision and convergence speed. *Last*, the escalating link bandwidth and relatively stagnant inter-host resources cause emerging host congestion [2, 3], necessitating accurate differentiation between the fabric and host congestion. Thus, CCs must be constantly updated to keep pace with the evolving DCNs.

2.1.2 Deploying New CC Algorithms is Complex. Deploying new CC algorithms at a large scale is typically complex as they are usually integrated into the datapath alongside the corresponding transport protocol stacks [22, 29, 44], necessitating laborious modification efforts as follows:

(i) Transport-specific Adaptations. CC algorithms require adaptations to meet the performance, security, and resource needs of the integrated transport. For hardware transports like RDMA [22], the design must consider hardware resource constraints (*e.g.*, connection scalability [69]), computational optimizations (*e.g.*, pipelined sorting[70]) and large-scale deployment (*e.g.*, avoiding deadlocks[45]). For software transports like kernel TCP [12], CC algorithms need to exclude the vulnerable points such as infinite loops and memory out-of-bounds access [7] to ensure kernel security and eliminate the time-consuming computation to meet stringent performance constraints [52].

(ii) Hardware Adjustments. Some algorithms depend on new hardware features and require hardware adjustments. For example, HPCC [36] mandates the incorporation of INT parsing and computation into RDMA NICs, whereas Bolt [9] relies on switch-initiated SRC feedback and modifies reactions. Some credit-based algorithms [14, 23, 51, 56] require the receiver to allocate credit. The ASIC NIC architecture architectures (*e.g.*, RNIC) typically evolve slowly compared with software stacks through iterative updates.

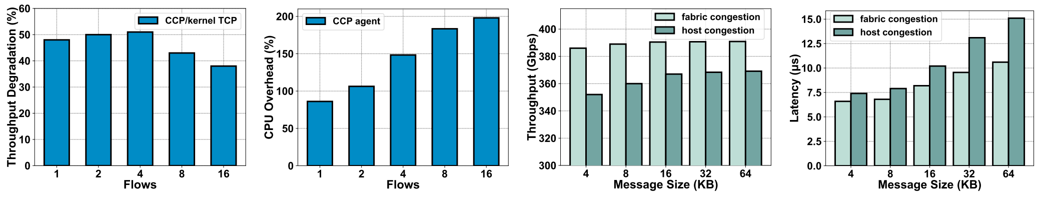
Hence, recent studies [15, 52] have been devoted to providing programmable CC frameworks with essential abstractions that help users to deploy custom-defined algorithms.

2.2 Desired Properties of CC Frameworks

Users desire a *high-performance* and *generic* CC framework:

(i) High-performance. The framework should be able to allow CC to achieve native performance comparable to in-datapath deployments. In hyper-speed DCNs (*e.g.*, 400 Gbps link speed), queue buildup can introduce tail latency on the order of tens of microseconds [9], and low link utilization can result in significantly increased flow completion times [36]. Therefore, the framework must ensure an accurate response to the congestion without introducing processing bottlenecks.

(ii) Generic. The framework needs to support diverse CC algorithms and congestion scenarios. On the one hand, the framework should accommodate different design trade-offs, including the congestion signal (*e.g.*, ECN [5], RTT [50], INT [24], etc.), decision period (*e.g.*, per-ack [36], per-RTT [5], etc.) and reaction mode (*e.g.*, window [33], rate [78], etc.). On the other hand, the framework should enable CC algorithms to handle different congestion scenarios, including host congestion [2] and fabric congestion [33].



(a) Throughput degradation under 400Gbps traffic. (b) High CPU overhead of the CCP agent under 400Gbps traffic. (c) Throughput reduction under host congestion in DOCA-PCC. (d) Latency increase under host congestion in DOCA-PCC.

Fig. 2. CCP has performance issues while DOCA-PCC lack generality for handling host congestion.

2.3 Existing Works & Limitations

Here, we introduce the primary architectures of existing works and analyze their performance and generality limitations.

CCP[52]. CCP leverages host CPUs to guarantee utmost programmability. It deploys an independent agent running in the user space to facilitate sophisticated CC algorithms (*e.g.*, Bayesian forecasts [72], online learning [16, 74], etc.). However, escalating traffic loads can exceed the agent's processing capacity, resulting in throughput degradation and overhead increases. Specifically, we deploy CCP on commodity servers equipped with Mellanox CX7 NICs operating at 400 Gbps links, and compare the throughput and CPU overhead when running the same CC algorithm in CCP and the native datapath (*i.e.*, kernel TCP). Experimental results are illustrated in Figure 2a and 2b. Compared to in-datapath deployment, CCP experiences throughput degradation ranging from 38% to 51% and introduces CPU overheads ranging from 86% to 198%. By analyzing the agent logs, we discover that the total throughput rises as concurrent flow increases. The escalating traffic loads lead to considerable CC event reports and frequent event drops, which compromise the speed and accuracy of congestion response.

DOCA-PCC[15]. DOCA-PCC employs programmable hardware (*e.g.*, BlueField [18]) to achieve flexibility while ensuring high performance. It offloads signal collection to the hardware (*e.g.*, DPA [17]) and performs decisions on the SoC. Although DOCA-PCC enhances CC response speeds, it falls short in addressing emerging host congestion in modern DCNs [2, 65]. *On the one hand*, diverse host signals [2, 3, 33] are required to locate different congestion locations and severity within hosts. However, DOCA-PCC offloads its signal collection into external hardware and only provides fabric and NIC signals. *On the other hand*, host congestion occurs between the CPU and peripherals, requiring the CPU to handle resource competition to mitigate congestion. In contrast, DOCA-PCC completely offloads decision-making to the SoC, which hinders the resolution of congestion issues at its core. As a demonstration, we deploy DOCA-PCC on BlueField-3 DPUs with its provided CC algorithm and evaluate its performance under different congestion scenarios. Similar to prior studies [3, 35], we employ the incast traffic representing network congestion and CPU-to-memory traffic representing host congestion (see §5.1 for detailed settings). Test results are shown in Figure 2c and 2d. Compared to fabric congestion scenarios, DOCA-PCC exhibits a 5.8% to 15.2% reduction in throughput and a 12.2% to 42.4% increase in latency under host congestion. Simply adjusting the sending rate to the network fails to address the root causes of host congestion, ultimately impairing overall performance.

Table 1 summarizes the focal points of existing works. Since they are unable to achieve both high performance and generality, we turn to designing a comprehensive solution.

Table 1. Existing works and their focal points. CCP relies on host software for signal collection and decision, suffering from performance degradation and significant CPU overhead. DOCA-PCC fully offloads CC tasks to external hardware, yet neglecting the detection and resolution of host congestion.

Existing Framework	**Signal Collection**	**Provided Signal**	**Decision Mode**
CCP[52]	Software-only	Fabric	CPU-hosted
DOCA-PCC[15]	Hardware-only	Fabric & NIC	SoC-hosted

3 Taurus Design

We first introduce the design rationale based on the framework's desired properties (§3.1). Next, we present the architecture of the hardware-software co-design framework (§3.2), discussing the primary challenges and our solutions (§3.3 - §3.5).

3.1 Design Rationale & Challenges

Taurus aims to provide a high-performance and generic CC framework for DCNs. As discussed and experimented in §2.3, we find that achieving high performance favors hardware assistance to handle heavy traffic workloads, while ensuring generality requires host software involvement to identify and address different congestion scenarios.

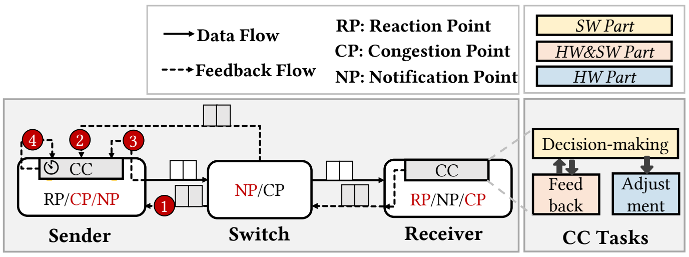


Fig. 3. The conceptual model of congestion control in modern DCNs and its expectations for task partitions.

To achieve the above goal, we partition CC functions into distinct tasks and map them onto suitable hardware/software components, while mitigating excessive communication overhead. We first construct a conceptual CC model widely employed in DCNs to describe CC tasks [5, 78]. Next, we analyze CC tasks in terms of their performance and flexibility requirements to determine the proper hardware/software mappings. As depicted in Figure 3, the sender injects packets into networks, the switch forwards packets, and the receiver generates feedback notifications. For the host side, the CC typically comprises three main tasks [15, 52] as follows:

- **Feedback Task.** The task involves generating and processing feedback notifications, which requires both *flexibility* and *high performance*. On the one hand, algorithms can have various feedback mechanisms, including ACK-based feedback (❶), switch-initiated feedback (❷), information from transmitted messages (❸), and timing-based feedback (❹). On the other hand, extracting congestion signals from high-speed datapaths (*e.g.*, 100 - 400Gbps link speeds [32]) requires a high processing capability (100+ Mpps).
- **Decision-making Task.** This task entails judging congestion and reacting accordingly, which is a focal point of CC *flexibility*. Firstly, different CC algorithms usually employ distinct judgment strategies, such as the switch queue length [5], RTT variation [50], link utilization [36], host IIO occupancy [3], etc. Secondly, the parameter setting and reaction mechanism of CC can vary depending on the application scenario (as discussed in §2.1). For performance concerns, previous large-scale CC deployments in DCNs [5, 9, 33] have demonstrated that the host CPU is sufficient for promptly carrying out CC decisions.

- **Adjustment Task.** This task determines the data path's delivery behavior in the next cycle, requiring it to be *high-performance*. For example, as mentioned in [70], RNICs can achieve speeds of up to 110 Mpps, and the rate-limiting adjustment needs to be quick enough to transmit a packet every 8 ns. Thus, the adjustment needs to be fast. In contrast, adjustment modes are typically fixed, such as the window-based [5] and the rate-based [78] scheme.

Therefore, we map these CC tasks to suitable devices, as shown on the right of Figure 3. The decision-making task is delegated to software, the adjustment task is partitioned to hardware, and the feedback task is collectively handled by both software and hardware. Under this architecture, the following three key design challenges should be addressed:

- **Challenges#1: Diverse Congestion Signals.** As discussed above, CC algorithms rely on different feedback signals that differ in type, source, and generation frequency, and may evolve over time as network conditions change. This diversity necessitates a feedback mechanism that is both flexible and extensible.
- **Challenges#2: Frequent Interaction Overhead.** The decoupled architecture introduces processing overhead during the exchange of feedback signals and decisions (see §3.4 for quantitative analysis). Therefore, an efficient interaction mechanism is required to minimize overheads.
- **Challenges#3: Seamless Algorithm Integration.** Users expect to deploy custom algorithms as easily as traditional software, without needing to manage signal conflicts, message overflows, or other runtime exceptions.

3.2 Framework Overview

To address the aforementioned challenges, we design Taurus, a high-performance and generic CC framework. As depicted in Figure 4, the framework consists of two primary components:

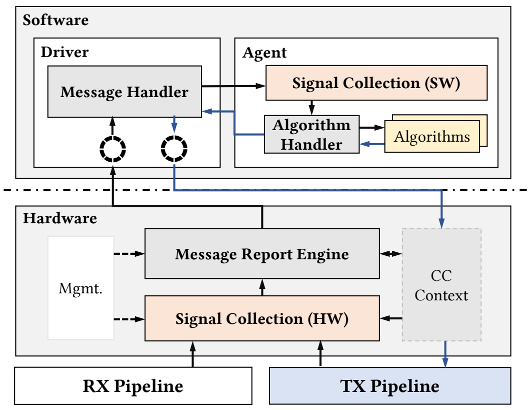


Fig. 4. Taurus architecture overview: Hardware collects congestion signals and reports them as structured messages; software interprets messages and host signals into CC events for algorithm execution.

(i) Hardware Part. The hardware part is responsible for extracting congestion signals and reporting messages to the software. To support diverse congestion signals, Taurus proposes a collaborative signal collection mechanism (§3.3). The hardware part tries its best to extract predefined congestion signals and custom feedback segments from the RX and TX pipeline, leaving the fine-grained collection of custom and host signals to the software. To reduce communication overhead due to frequent signal reporting, Taurus designs a type-aware message report engine that performs two-tier aggregation for native signals (§3.4), generating different messages to the software.

(ii) Software Part. The software part includes a driver for managing message reporting and an agent for deploying algorithms (§3.5). First, the message handler fetches messages and manages exceptions such as overflows and conflicts. Subsequently, messages are sent to the software signal collection module, where custom and host signals are collected together. Next, the algorithm handler generates distinct events to notify the corresponding algorithms. After CC decisions are made, the algorithm handler checks results and removes constants or out-of-range values to prevent unnecessary updates to the hardware.

Feasibility Discussion. The separate pattern intuitively raises concerns about the effectiveness of CC. From a practical standpoint, the separation indeed delays feedback, whereas the introduced latency between hardware and software is relatively minor (*e.g.*, 1us PCIe latency) compared to the overall end-to-end delay, especially in congested scenarios (tens of microseconds [5, 78]). We will provide quantitative validation through practical experiments (§5).

3.3 Collaborative Signal Collection

This component aims to extract congestion signals from the feedback. To reduce the burden of software extraction while allowing customized and host signal collection, we introduce a collaborative collection mechanism. The core idea is to harness the hardware's ability for best-effort signal extraction while leaving fine-grained signal analysis to the software.

In detail, the signal collection module categorizes congestion signals into three types: (i) *predefined* signals that are widely used (*e.g.*, ECE, ACK, etc.); (ii) *custom* signals defined by users (*e.g.*, INT, credit, etc.); (iii) *host* signals reflecting host network status (*e.g.*, IIO occupancy, etc.). Accordingly, hardware is used to obtain *predefined* signals and feedback segments containing *custom* signals; while software is used to obtain specific *custom* signals and *host* signals. Figure 5 illustrates the collaborative signal collection overview, which consists of two parts:

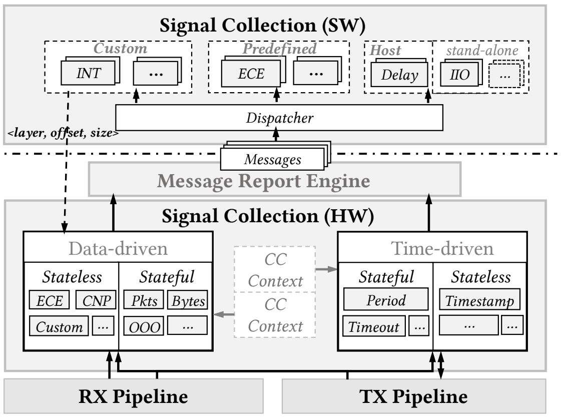


Fig. 5. Collaborative signal collection, where the hardware part tries its best to extract signals, while the software part further obtains custom and host signals.

Signal collection in hardware. The hardware collection module provides two signal extraction approaches based on their triggering modes: *data-driven* and *time-driven.*

- *Data-driven.* These signals originate from the received packets from the RX pipeline and data sent to the TX pipeline. Modules along the pipeline are interconnected with the collection module to provide metadata or the entire payloads. For stateless signals, such as ECE [5] and CNP [78], Taurus directly forwards them to the message report module. For stateful signals (*e.g.*, out-of-order), Taurus stores them in the CC context with the corresponding flow IDs to determine the actual state. In particular, users can define custom signals beyond the standard protocols, making it challenging for hardware to extract them flexibly. Therefore, Taurus provides a selective extraction method. The software issues configurations to enable hardware to extract specific segments from the given location, avoiding directly sending the entire feedback payload to the software. Afterward, the extracted segments are transmitted to the software for further fine-grained parsing.

- *Time-driven.* These signals are triggered by hardware timers, providing different feedback schemes. Taurus supports stateless (*e.g.*, timestamps [50]) and stateful signal (*e.g.*, timeouts [33] and periods [78]) collection. To prevent conflicts when accessing CC contexts simultaneously with data-driven signals, Taurus keeps the CC contexts of the two independent from each other. Taurus provides a proactive signal collection function that sends probing packets to obtain link delays as required by some algorithms [42, 43]. Currently, Taurus provides two methods: one is to attach the probe request into the transmit data payload at fixed intervals of N packets or T time units; the other is to immediately generate independent probe packets upon receiving software-initiated commands. The probe field consists of 8 bytes, including the mode to indicate packet type, a timestamp for the record, and a probe sequence number to avoid time reversal.

Signal collection in software. The signals collected by the hardware are fed into the message reporting engine (§3.4) where they are turned into messages along with signal types and arrival timestamps. Messages are sent to the software and dispatched to different collection modules based on the signal types. *Pre-defined* signals can be directly obtained without additional operations since they have already been extracted by the hardware. *Custom* signals need to be parsed from the feedback segment based on the user-defined structures (*e.g.*, INT[24], credit [56]). *Host* signals are collected in two distinct ways: a portion of host signals are embedded into the remote feedback packets (*e.g.*, remote endpoint delay [33]), while the other is independently sampled from the local host (*e.g.*, IIO occupancy [3]). For the former, Taurus extracts signals from the feedback data segments using the same approach as custom signals. For the latter, Taurus presents a stand-alone signal sampling method. In detail, Taurus modularizes the sampling functions and dynamically adds new functions based on algorithm needs. Each sampling function can be added and called independently. For example, the MSR sampling [25] is added to obtain the IIO occupancy and PCIe bandwidth to pinpoint the severity of host congestion. VTune Profiler [27] is used to obtain the host CPU utilization to identify the thread activity of applications.

3.4 Type-aware Message Report Engine

Compared to in-datapath CC deployment [12, 78], the framework introduces additional communication between the hardware and software. This raises two critical issues: (i) Per-packet/interval feedback can consume considerable shared host resources (*e.g.*, PCIe bandwidth), resulting in excessive communication overheads. (ii) Small-sized feedback signals (*e.g.*, 1-bit ECN) reduce communication efficiency. Specifically, the theoretical bandwidth consumption (BW) and communication efficiency (E) for the feedback (f_i) can be formulated as follows:

$$BW = \sum_{f_i} (F(f_i) \times S(f_i)) \tag{1}$$

$$E = Avg(\sum_{f_i} (S(f_i)/(S(f_i) + S(H)))) \tag{2}$$

Where f_i refers to the index of feedback, $F(f_i)$ denotes the feedback frequency, $S(f_i)$ indicates feedback data size, and $S(H)$ signifies extra packet header size in transmissions.

In Equation 1, the value of $F(f_i)$ varies depending on the type of feedback. For time-driven signals, $F(f_i)$ is inversely proportional to the signal generation period, which is typically in the microsecond range (*e.g.*, RTT [50], TX intervals [78], etc.). Thus, $F(f_i)$ typically ranges from 100*s* Krps to 1*s* Mrps. For data-driven signals, $F(f_i)$ is related to the frequency of data sending or feedback reception. In the case of the typical ACK feedback, $F(f_i)$ can reach up to 12.5-50 Mrps at a link rate of 400Gbps with 4K/1K MTUs. Consequently, such a high feedback rate consumes substantial PCIe bandwidth and burdens the host processing. In Equation 2, the E depends on the ratio between feedback size $S(f)$ and the header size $S(H)$ in PCIe transmission (*e.g.*, 4DW TLP header [63]). Thus, for small-sized feedback signals (*e.g.*, 1-bit ECN flag), the headers occupy a significant portion of the transmission bandwidth, leading to reduced communication efficiency in practice.

To address these concerns, TAURUS presents a type-aware message report engine. As depicted in Figure 6, it employs a two-tier aggregation mechanism to reduce communication overhead:

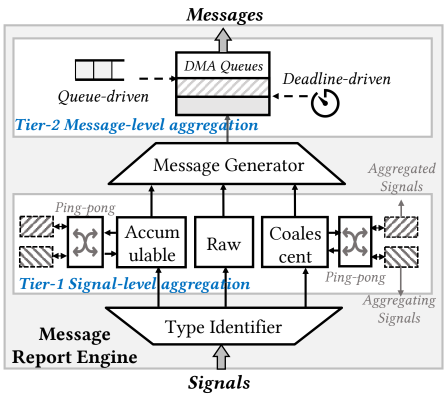
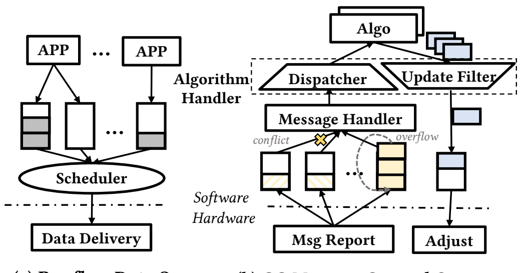


Fig. 6. Type-aware message report engine. TAURUS aggregates different types of signals and messages to reduce communication overhead and improve efficiency.

Signal-level Aggregation. The purpose of this aggregation is to reduce the frequency of feedback reporting, $F(f_i)$. In detail, the signals are sent to three independent processing engines based on their aggregated capabilities: *Accumulable*, *Coalescent*, and *Raw*. The *Accumulable* engine sums up the input signals to calculate the total quantity, such as the amount of received ACK and ECE [5], and the number of bytes and packets sent [78]. The *Coalescent* engine merges multiple signals over a period of time, such as out-of-order and packet loss signals [5]. Since these signals often recur before adjustment takes effect during congestion, signal coalescence can avoid over-control. The *Raw* engine allows direct forwarding of non-aggregated signals and custom signals. Each aggregation engine has an independent aggregation period, configured by algorithms. All signals are then sent to the message generator. TAURUS designs a ping-pong buffer [31] to separate signals currently being aggregated from those that have been aggregated to avoid read/write conflict.

Message-level Aggregation. This module aims to improve communication efficiency E by increasing the feedback size per transmission. Specifically, TAURUS introduces two mechanisms for message aggregation: *queue-driven* and *deadline-driven*. Messages pushed into the DMA queue are not sent immediately. They are reported to the software in batches only when the cumulative queue length surpasses the queue threshold. Generally, a shallow queue depth is often sufficient to achieve high efficiency for this queue (*e.g.*, 128~512B [63]). However, messages may experience long waiting times when lacking enough feedback in some cases. To guarantee timely reporting, the deadline-driven mechanism initiates message reporting if the queuing times exceed the deadline.



(a) Per-flow Data Queue (b) CC Message Control Queue

Fig. 7. The diagram of the communication between hardware and software, along with the software handler.

3.5 Software Built-in Handler

Taurus employs the ring buffer queues [58, 73] for communication, as illustrated in Figure 7. In the data plane, tens of thousands of queues are deployed and scheduled [69] to fulfill the requirements of isolation [67] and QoS [33] for applications. In the control plane, maintaining numerous control queues for CC is unnecessary, as it aims to achieve fairness for each flow [33]. Hence, Taurus constructs message-based control queues instead of per-flow queues to reduce scheduling overhead.

To facilitate algorithm deployment, Taurus provides software handlers to address three practical issues. First, the message handler copes with exceptional reporting, such as message overflows and conflicts. Secondly, the algorithm handler provides only the subset of the signal that algorithms truly require, rather than all feedback. Additionally, it removes the non-essential updates that fall outside reasonable ranges or unchanged values.

Message Handler. This component is responsible for managing exception reports. On the one hand, when the software processing rate falls behind the hardware reporting rate due to jitters [33], new messages may overwrite unfetched messages due to the ring buffer overflow. To address this issue, Taurus utilizes the back-pressure approach to decrease the reporting rate by proactively increasing the aggregation period in the message report engine. For unmergeable signals, one practical strategy is to evict older entries in the ring buffer to accommodate new signals. Future work may explore more fine-grained approaches, such as selective eviction policies. On the other hand, the order of message receiving in software may differ from the order of feedback arrivals in hardware due to aggregations. To preserve temporal information, Taurus records the initial arrival time of aggregated signals in the hardware and the message reporting timestamps in the software to determine the chronological order of signal occurrences and avoid erroneous judgments.

Algorithm Handler. The agent receives messages through the memory mapping approach and generates feedback events to CC algorithms. Given that algorithms only require a subset of signals for decision-making, Taurus provides interfaces for users to bind their algorithms with the required feedback events (§4.2). Therefore, users do not have to deal with unnecessary feedback signals. Furthermore, Taurus incorporates the update filter to mitigate unnecessary updates to the hardware. Currently, Taurus provides two filters: *boundary filter* and *duplicate value filter*. The boundary filter limits the decision results within the boundaries of data transfer capabilities (*e.g.*, 100 Kbps to 400 Gbps). The duplicate value filter intercepts updates if results remain consistent with previous values. The hardware maintains the previous decisions in the absence of updates.

4 Implementation

We have implemented a fully functional Taurus with FPGA-based NICs on commodity servers and deployed it into DCNs operating at 400Gbps link speeds.

4.1 Hardware Integration

We use the FPGA-based NIC as a prototype to validate the feasibility of Taurus. Specifically, we select a custom FPGA board with Intel® Agilex™ chips [11], equipped with a 400G QSFP port and a PCIe Gen5x16 interface. We implement both **basic NIC** functions and the in-house designed **RDMA** transport to represent high-performance datapaths.

CC Module. We implement the hardware logic described in §3 using System Verilog. The operational clock frequency is configured at 300 MHz. The CC module consumes 1.9% of ALMs, 1.43% of Registers, 1.27% of M2Ks, and 0.78% of BRAMs. The resource overhead is minimal, with the majority of the storage resource utilization stemming from the CC context.

Other NIC Modules. The CC module interconnects with modules on the NIC's RX and TX pipelines to obtain metadata or packet payloads from them. In our implementation, the CC module gains signals from L3-L4 network modules (*e.g.*, IP, RDMA, etc.). Additionally, the CC adjustment should be timely and scalable. Apart from the typical window-based control mode, Taurus realizes a fast and accurate rate limiter based on a timing wheel with the WF^2Q^+ algorithm [10] that supports 10K+ flows to regulate their rates.

4.2 CC Algorithm Deployment

We develop the CC agent using the C++ language. It runs in the user space to uniformly manage CC executions. Users can easily deploy CC algorithms using Taurus's APIs (detailed APIs and examples in the §A). Specifically, implementing a CC algorithm using Taurus involves four stages:

(1) **CC Initialization.** Users allocate memory for CC contexts, initialize parameters, and configure states based on their deployment scenarios (slow start or maximum rate start).
(2) **Event Binding.** Users bind the desired CC events from all feedback events as inputs to their custom decision logic.
(3) **Decision Execution.** The CC algorithm evaluates the congestion status and executes appropriate reaction strategies. For instance, the algorithm can adjust the sending rate to the fabric to avoid network congestion [5, 78] or invoke host resource management tools (*e.g.*, MBA [61], Memory QoS [6], etc.) to deal with resource contention and mitigate host congestion.
(4) **Result Update.** After making CC decisions, users write the results back to the hardware CC context for use in the next adjustment cycle.

5 Evaluation

In this section, we conduct testbed experiments along with large-scale NS3 simulations to evaluate Taurus's performance by answering the following questions:

1. Are the component designs of Taurus effective? We employ a series of micro-benchmarks to demonstrate that Taurus can alleviate communication overhead by 14.2% to 95.9% and CPU overhead by 23.5% to 58.1%(§5.2).

2. Can Taurus provide a high performance and generality CC framework? We evaluate the system performance with representative CC algorithms and distinct congestion scenarios, demonstrating that Taurus can uphold high performance across diverse metrics (§5.3).

3. How does Taurus compare against related frameworks? We compare Taurus with software CCP and hardware DOCA-PCC, showing that Taurus improves throughput by 32.3%, reduces latency by 96.4% and lowers CPU overhead by 158.7%, compared to CCP. In comparison with DOCA-PCC, Taurus improves throughput by 9.3% and reduces latency by 28.8% (§5.4).

5.1 Evaluation Setup

Network Topologies. We set up the testbed and large-scale NS3 simulation [55] to evaluate Taurus. The testbed represents the typical DCN pod, comprising two Agg switches and two ToR switches. Each ToR is connected to eight servers. The base RTT between ToR switches is $5\mu s$ and $8\mu s$ across racks. The MTU is configured to 4K. A server is equipped with two FPGAs and BlueField-3 B3140H DPUs, two 8-core Intel® Xeon® 8457C CPUs and 512GB of memory.

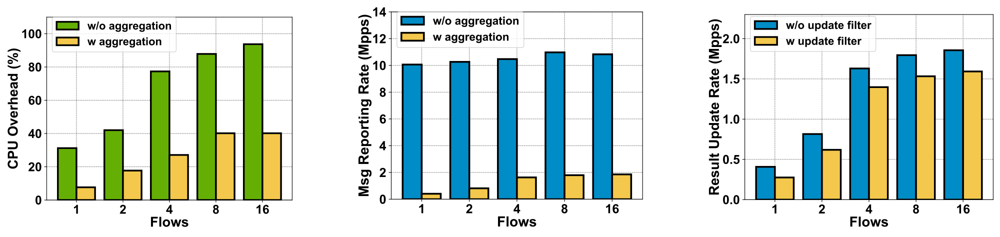
The NS3 simulation topology is based on a typical 3-layer FatTree [4], comprising 16 Core switches, 20 Aggregation switches, and 20 ToRs switches, without over-subscription. 320 servers are evenly located under ToRs. Each link is 400 Gbps and the propagation latency is set to $1\mu s$. All switches support PFC and set 32MB shared buffer. We supplement the PCIe access model described in [54] into the simulation to mimic the Taurus separation architecture.

Traffic Patterns. We evaluate Taurus using two realistic datacenter workload patterns, *Web-search* [78] and *Hadoop* [59], for end-to-end FCT. In addition, we use synthetic flows based on prior industry work as micro-benchmarks. To evaluate performance under different congestion scenarios, we generate incast traffic by deploying multiple senders to a single receiver (*i.e.*, 8:1), following prior work [5, 36], and employ MLC [26] to produce CPU-to-memory traffic to represent the local memory-intensive applications indicating host congestion [3]. This traffic is used for comparing different frameworks. Tool details can be found in the appendix.

CC Algorithms and Frameworks. We select a wide range of CC algorithms that have been widely deployed at scale in DCNs for evaluation, including DCTCP [5], Swift [33], HPCC [36], and Bolt [9]. These algorithms exhibit distinct congestion signals and decision strategies, representing ECN-based, delay-based, INT-based, and queue-based algorithms, respectively. We configure parameters based on the algorithm recommendations. Their native in-datapath implementations are chosen as the baseline[1]. For the CC framework, we compare Taurus with both CCP [52] and DOCA-PCC [15] in terms of their performance and CPU overhead.

5.2 Micro-benchmarks

We use micro-benchmarks to evaluate the effectiveness of component designs in Taurus, including the aggregation mechanism in hardware and the update filter in software handlers. We compare the CPU and communication overhead with and without aggregation, as well as the result update rate with and without update filtering.



(a) CPU overhead under 400Gbps traffic (Testbed).

(b) Message rate under 400Gbps traffic (Testbed).

(c) Result update rate under 400Gbps traffic (Testbed).

Fig. 8. Taurus reduces CPU overhead and message reporting rate in communication with the aggregation and alleviates unnecessary update overhead with the update filter.

[1]Since Swift, HPCC, and Bolt are tightly integrated with proprietary transport datapaths, we compare them in the simulation and deploy the internal DCTCP algorithm (denoted as iDCTCP) in the testbed. iDCTCP uses ECE, RTO, and OOO as feedback with the same AIMD response mode as DCTCP.

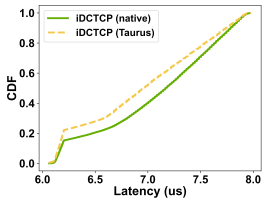
Taurus's aggregation reduces the CPU and communication overhead. We measure the CPU overhead and the message reporting rate during hardware-software communication under traffic loads of 400 Gbps. The CPU overhead is quantified by CPU utilization, with the maximum value depending on the total number of available CPU cores (*e.g.*, 4800% for a 48-core server). Figure 8a and 8b show the results. Without aggregation, each feedback triggers a message report, which consumes CPU resources up to 98.1% and achieves a report rate of 10.8 Mpps. In contrast, using aggregation can significantly reduce the message reporting rate by 82.8% to 95.9% and reduce CPU overhead by 23.5% to 58.1%. Taurus consumes only a portion of a single core's resources to tackle 400 Gbps traffic feedback, making it well-suited for commercial multi-core servers (e.g., typically equipped with 96–128 cores [28]).

Taurus's update filter reduces the unnecessary update overhead. We compare update rates with and without the filter while handling the same traffic. The results are shown in Figure 8c. The update filter reduces the update rate from 14.2% to 32.5%. This reduction occurs because CC algorithms have different control periods for different congestion statuses (*e.g.*, per-packet increase [36], per-RTT decrease [33]). Some feedback signals remain unchanged beyond the decision period. Therefore, the update filter helps to avoid unnecessary update overhead.

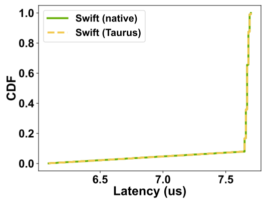
5.3 Taurus Performance

The goal of Taurus is to provide a high-performance and generic CC framework. To evaluate performance, we test Taurus using various metrics such as throughput, latency, queue usage, and FCT. To evaluate generality, we compare a variety of CC algorithms and congestion scenarios.

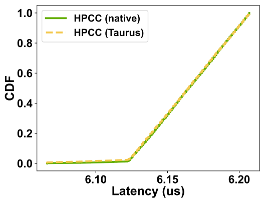
Traffic Latency. We measure the traffic latency of 10K mice flows that traverse through a link saturated by the two elephant flows, similar to the study in [36]. The experimental results are shown in Figure 9a-9d. Taurus exhibits latency distributions similar to those of the native in-datapath CC. Surprisingly, Taurus even slightly reduces the latency of the mice flows. Further analysis reveals that the delay caused by separating the flows slightly slows down the rate of increase, especially for conservative per-RTT additive increase. Thus, before the elephant flow can fully saturate the link bandwidth, the mouse flow has already acquired available bandwidth to complete its transmission.



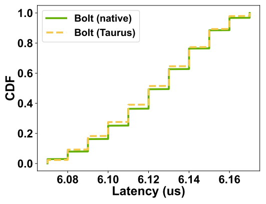
(a) ECN-based algorithm (iDCTCP, Testbed).



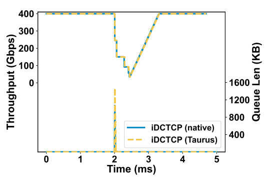
(b) Delay-based algorithm (Swift, Sim).



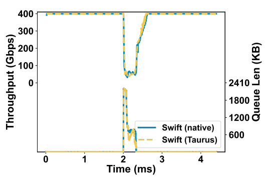
(c) INT-based algorithm (HPCC, Sim).



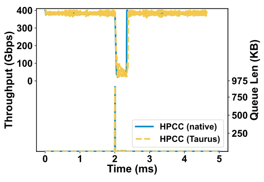
(d) Queue-based algorithm (Bolt, Sim).



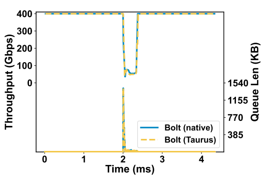
(e) ECN-based algorithm (iDCTCP, Testbed).



(f) Delay-based algorithm (Swift, Sim).



(g) INT-based algorithm (HPCC, Sim).



(h) Queue-based algorithm (Bolt, Sim).

Fig. 9. The performance comparison of CC algorithms using Taurus and native implementation.

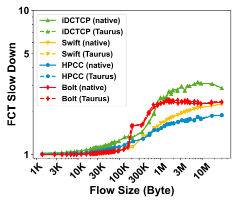


Fig. 10. Avg FCT slowdown under *WebSearch* traffic (Sim).

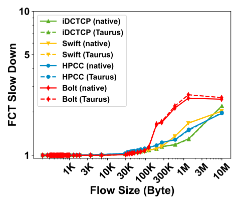


Fig. 11. Avg FCT slowdown under *Hadoop* traffic (Sim).

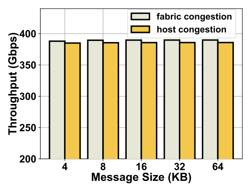


Fig. 12. Throughput under distinct congestion scenarios (Testbed).

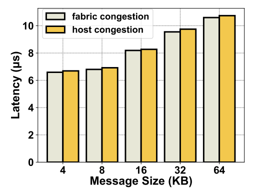


Fig. 13. Latency under distinct congestion scenarios latency (Testbed).

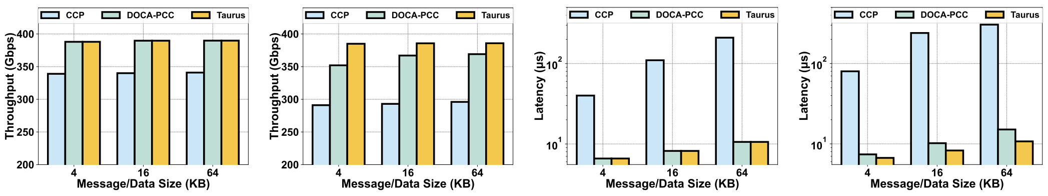
Throughput & Queue Usage. We further evaluate the impact on overall throughput and switch queue usage when introducing a new flow into a congested link. Specifically, one senders saturate the link and other seven senders injects 500 KB flow at 2 ms. As depicted in Figure 9e-9h, different algorithms exhibit varying effects on the convergence rate of flow throughput. All algorithms can eventually converge to the full bandwidth. Taurus introduces a small degree of fluctuation in the process of convergence for Swift, HPCC and Bolt algorithms, which is mainly due to their latency-sensitive reactions or fast feedback periods (less than one RTT). For the iDCTCP algorithms, Taurus exhibits a minor impact on throughput. Meanwhile, we monitor queue usage from the switch buffer and find that it behaves similarly to throughput performance. Overall, Taurus does not cause notable penalties at the granularity of RTT.

FCT Slowdown. We measure the flow completion time (FCT) of selected CC algorithms under the *WebSearch* and *Hadoop* traffic. We compare the actual FCT with the ideal FCT to get the FCT slowdown. To stress the test, we introduce the incast background traffic with a flow size of around 500KB, as done in [36]. The experimental results are represented in Figure 14. The FCT slowdown exhibits differences in both CC algorithms and traffic patterns. Overall, Taurus achieves near-native FCT slowdown. For small-size flows (shorter than 100KB), the additional latency introduced by Taurus slightly increases FCT by an average of approximately 0.2%. For large-size flows (larger than 300KB), the separation delay in Taurus slightly slows down their ramp-up, thereby marginally increasing the FCT slowdown for some algorithms.

Algorithm-wise & Congestion-wise Generality. Taurus provides a variety of congestion signals (§3) for CC algorithms. Figure 9 illustrates Taurus's high performance across different algorithms, demonstrating its algorithm-wise generality. Additionally, we compare the throughput and latency under both host congestion and network congestion scenarios (settings are described in §5.1). We deploy the iDCTCP algorithm and integrate a recently proposed HostCC approach[3], which mitigates host congestion by applying backpressure mechanisms to local CPU-to-memory traffic. The experimental results are presented in Figure 12 and 13. Taurus achieves high throughput and low latency in both situations. This is primarily because Taurus introduces the collaborative signal collection mechanism capable of providing both network and host congestion signals for users to differentiate congestion points and take appropriate actions. In summary, Taurus can achieve algorithm-wise and congestion-wise generality.

5.4 Taurus vs. Existing Frameworks

We now compare Taurus with existing frameworks (CCP[52] and DOCA-PCC[15]) in terms of their throughput, latency, and CPU overhead under different congestion scenarios. We deploy each framework on the same server and integrate it with the transport datapaths it supports. Currently, the CCP framework only supports software datapaths and we choose the widely used kernel TCP datapath. DOCA-PCC supports hardware transport paths and we choose the RDMA datapaths. Taurus is deployed as outline in §4.



(a) Throughput in fabric congestion (Testbed).
(b) Throughput in host congestion (Testbed).
(c) Avg latency in fabric congestion (Testbed).
(d) Avg latency in host congestion (Testbed).

Fig. 14. Performance comparison of Taurus with other CC frameworks in different congestion scenarios.

Throughput. We measure the throughput of each framework with different data sizes under both network congestion and fabric congestion scenarios. The results are shown in Figure 14a and 14b. In scenarios with fabric congestion, both Taurus and DOCA-PCC can achieve extremely high throughput, reaching around 390 Gbps. In contrast, CCP's throughput is limited to around 340 Gbps. This is because both DOCA-PCC and Taurus leverage hardware to execute performance-sensitive CC tasks, thereby avoiding processing bottlenecks. However, CCP's software processing capabilities struggle to address the heavy traffic. The agent logs indicate considerable CC event reports and frequent CC event drops, impacting the timeliness and accuracy of congestion response and finally affecting the CC algorithm to reach the desired convergence throughput. In situations with host congestion, DOCA-PCC sees a notable drop in throughput. This is because it does not support host signal collection, which is important to identify the locations and severity of host congestion. Moreover, simply adjusting the sending rate can not fundamentally address the root cause of host congestion. In contrast, Taurus enhances throughput by 9.3% by utilizing software for congestion signal collection and performing host CC algorithms.

Latency. We measure the average latency for different data sizes under different congestion scenarios. The results are depicted in Figure 14c and 14d. In the case of fabric congestion, DOCA-PCC and Taurus show very low latency, ranging from 5 to 10 μs across different data sizes. However, CCP experiences higher latency (40-210μs) compared to them. There are two main reasons for this difference. First, CCP operates with the software transport datapath, which generally has higher latency compared to the hardware transport datapath [21]. Second, due to the reasons mentioned earlier, CCP's poor response to congestion can lead to long queue delays. In the scenarios with host congestion, DOCA-PCC exhibits the latency increase, while Taurus demonstrates a 28.8% reduction in average latency compared to DOCA-PCC. This is similar to the results and reasons seen in the throughput experiments. Overall, Taurus exhibits more adaptability to different scenarios.

CPU Overhead. We gather the host CPU utilization of each framework to evaluate their CPU overhead. The experimental results are presented in Table 2. The CPU overhead of CCP ranges from 86.1% to 198.7%. In comparison, Taurus's CPU overhead is 7.8% to 40%, reducing 78.3% - 158.7% overheads. This improvement is primarily attributed to two factors. Firstly, Taurus offloads performance-sensitive tasks to hardware, thereby reducing the software workloads. Secondly, the hierarchical aggregation scheme helps reduce frequent data reporting, thus alleviating the CPU processing burden. Considering the 400Gbps traffic loads and the prevalent use of multi-core CPUs in commercial servers [28], the overhead of Taurus is deemed acceptable. Since DOCA-PCC offloads its decision-making part to hardware, it does not consume host CPU cores. However, as previously discussed, it misses the opportunity to collect host signals and coordinate host resources, which is significant for CC algorithms.

Table 2. The host CPU overhead among different frameworks. *DOCA-PCC runs in the hardware.

Flows	1	2	4	8	16	32
CCP	86.1%	106.3%	148.3%	183.4%	198.0%	198.7%
TAURUS	7.8%	17.7%	27.1%	40.0%	40.0%	40.0%

6 Discussion

Receiver-driven CC. Taurus aims to support send-driven CC schemes widely deployed in modern datacenters (*e.g.*, Google [33], Alibaba [36]), Bytedance [69], etc.), recent studies propose receiver-driven CC (*e.g.* Homa [51], EQDS [56], SRID [57], etc.), which involve the receiver proactively adjusting the sender's rate to alleviate the incast congestion and reduce tail latency. However, receiver-driven CC requires an additional RTT to convey credit and only captures last-hop congestion, which has not been widely adopted in production. Extending Taurus to support it would first require identifying receiver feedback signals, such as GRANT packets in Homa [51] or PULL packets in EQDS [56], which can be treated as *custom feedback*. The next step would be to characterize the performance requirements of CC tasks. We leave their implementation for future work.

7 Experience and Lessons Learnt

Taurus has been deployed in production datacenter networks for several years. We summarize key takeaways from the deployment experience:

Programmable CC is essential for hardware transport. Over the past few years, major cloud providers have offloaded network transport onto hardware to meet stringent performance demands. For example, Google introduced Falcon [60] for AI and HPC, and Alibaba designed LUNA [77] and SOLAR [49] for storage clusters. A key challenge lies in enabling these fixed hardware stacks to deliver optimal performance across diverse workloads and traffic patterns at scale. We find that providing a programmable framework not only facilitates rapid validation of new CC algorithms for next-generation hardware, but also allows for deploying different CC tailored to different networks regions (*e.g.*, scale-up vs. scale-out networks [37]).

Message aggregation and update filtering matter. Throughout the production deployment, we observed bursty traffic patterns that caused transient spikes in message rates (from several Mpps up to tens of Mpps). Message aggregation was critical to reducing instantaneous processing load and avoiding CC signal loss. This finding is also demonstrated in CCP experiments §2.3. Moreover, redundant updates or values beyond data-plane capability have no practical effect; filtering them before dispatch significantly reduces host-to-device traffic to reduce bus resource competition.

HW/SW co-design is worth further exploration. Early designs employ either software-only or hardware-only solutions to maximize flexibility or performance. Later studies embedded ARM [47] or RISC-V cores [17] to balance these tradeoffs. In TAURUS, we employ HW/SW co-design to further enhance generality. Looking ahead, as scale-out networks advance toward 800–1600 Gbps with hundreds of available paths [41], combining heterogeneous software resources (*e.g.*, ARM, RISC-V, host CPUs) to build hierarchical decision models for CC remains a promising direction.

8 Related work

ECN-based CC. Such algorithms [5, 42, 64, 78] leverage the widely deployed ECN mechanism in commercial switches as congestion indicators. Since the ECN is typically carried in the form of standardized packets such as ECE [5] or CNP [78], TAURUS extracts them as *pre-defined* signals.

Notably, ECE is carried on ACK packets and thus arrives at a higher frequency (§3.4). To alleviate software agent pressure, TAURUS aggregates these signals in the *Accumulable* engine before dispatching them to the software.

Delay-based CC. Such algorithms utilize variations in end-to-end delay [50, 76] or deviations from a target delay [33] as congestion signals. Multi-bit delay encodes the extent of congestion, not just its presence [33]. Since the delay is carried in ACK packets or probe response packets, TAURUS extracts them as *pre-defined* signals, reporting both fabric and end-host delay to the software for decision making. The RTT probing interval affects the frequency of signal reporting and processing, so it should avoid overly frequent probing that could impose software overhead.

INT-based CC. These algorithms leverage the emerging in-network telemetry technique [24] to obtain detailed in-network information (*e.g.*, actual rates) for more accurate decisions [36, 68]. INT information is embedded in packet headers defined by different protocols. TAURUS extracts the feedback segment containing the INT data from packets and delegates detailed parsing to the software. Since INT packets may carry multi-hop information, algorithms impose different bandwidth overheads and CPU processing costs. A promising future direction is to offload part of the INT parsing to the on-chip SoC, enabling pre-filtering before passing data to the software.

Learning-based CC. Recent work explores learning-based algorithms [16, 38, 46] that employ machine learning to adapt CC under diverse network conditions. It primarily targets wide-area Internet environments [38], where decision intervals are relatively long, and have not yet been deployed at scale in DCNs. As such, they remain outside the design scope of TAURUS.

CC Framework. Regarding signal collection, TAURUS identifies it as a task that is both performance-sensitive and flexibility-sensitive. Therefore, TAURUS adopts hardware-software collaborative collection as detailed in §3.3. For provided signals, TAURUS offers a wider range of congestion signals to cover more congestion scenarios in §4.2. In terms of CC decisions, TAURUS employs host software involvement to address congestion. We hope that our efforts can inspire existing frameworks to make further enhancements in the above aspects.

Other programmable platforms. Recent datacenter infrastructures integrate heterogeneous programmable devices, including programmable switches [75], FPGAs [34, 66], DPUs/SmartNICs [18, 20], and DPDK-based servers [19]. Programmable switches provide pipeline programmability with ultra-high throughput, parallel to NIC-based platforms. FPGAs and DPUs offload transport and network functions, with frameworks such as Tonic [8] and DOCA-PCC [15] supporting programmable CC, though limited in capturing host congestion. DPDK-based servers ensure high throughput in software but incur substantial CPU overhead. Compared with them, TAURUS adopts the HW/SW co-design architecture to balance performance, generality and flexibility.

9 Conclusion

There is a growing interest in both academia and industry in programmable congestion control (CC) frameworks. However, existing works either introduce performance issues or lack sufficient generality. This paper presents TAURUS, a hardware-software co-design CC framework to address this issue. TAURUS maps CC tasks into suitable components and proposes efficient mechanisms to reduce overheads. A large amount of experiments with real-world workloads demonstrate its performance and generality. We hope this framework can inspire more innovative CC mechanisms.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (Grant No. 62472404). Heng Pan is the corresponding author.

References

[1] Dennis Abts, Garrin Kimmell, Andrew Ling, John Kim, Matt Boyd, Andrew Bitar, Sahil Parmar, Ibrahim Ahmed, Roberto DiCecco, David Han, et al. A software-defined tensor streaming multiprocessor for large-scale machine learning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 567–580, 2022.

[2] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, et al. Understanding host interconnect congestion. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 198–204, 2022.

[3] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. Host congestion control. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 275–287, 2023.

[4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 38(4):63–74, 2008.

[5] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proc. SIGCOMM*, 2010.

[6] AMD64 Technology Platform Quality of Service Extensions. https://developer.amd.com/wp-content/resources/56375.pdf, 2020.

[7] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 97–112, 2018.

[8] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed nics. In *Proc. NSDI*, 2020.

[9] Serhat Arslan, Yuliang Li, Gautam Kumar, and Nandita Dukkipati. Bolt:sub-rtt congestion control for ultra-low latency. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 219–236, 2023.

[10] Jon CR Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on networking*, 5(5):675–689, 1997.

[11] Agilex™ FPGA Portfolio Product Brief. https://www.intel.com/content/www/us/en/content-details/758440/agilex-fpga-portfolio-product-brief.html, 2023.

[12] Vinton Cerf and Robert Kahn. A protocol for packet network intercommunication. *IEEE Transactions on communications*, 22(5):637–648, 1974.

[13] Hongtao Chen, Mingxing Zhang, Ke Yang, Kang Chen, Albert Zomaya, Yongwei Wu, and Xuehai Qian. Achieving sub-second pairwise query over evolving graphs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1–15, 2023.

[14] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 239–252, 2017.

[15] DOCA PCC Implementation On Top of NVIDIA® BlueField® networking platform. https://docs.nvidia.com/doca/sdk/nvidia+doca+pcc+application+guide/index.html, 2023.

[16] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. Pcc vivace:online-learning congestion control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 343–356, 2018.

[17] DPA Subsystem, DOCA Documentation v2.8.0. https://docs.nvidia.com/doca/sdk/dpa+subsystem/index.html, 2024.

[18] NVIDIA BlueField-3 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf, 2021.

[19] F-Stack: Open Source High Performant Network Framework based on DPDK. https://github.com/F-Stack/f-stack, 2017.

[20] Qiaoyin Gan, Heng Pan, Luyang Li, Kai Lv, Hongtao Guan, Zhaohua Wang, Zhenyu Li, and Gaogang Xie. Snary: A high-performance and generic smartnic-accelerated retrieval system. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pages 381–398, 2025.

[21] Chuanxiong Guo. Rdma in data centers: Looking back and looking forward. *Keynote at APNet*, 2017.

[22] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proc. SIGCOMM*, 2016.

[23] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 29–42, 2017.

[24] In-band Telemetry. https://docs.broadcom.com/doc/IBT-PB100, 2017.

[25] Intel® 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2023.

[26] Intel® Memory Latency Checker. (2023). https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html, 2023.

[27] Intel® VTune™ Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.gkpp99, 2024.

[28] Intel® Xeon® Processors. https://www.intel.com/content/www/us/en/products/details/processors/xeon.html, 2024.

[29] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level tcp stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, 2014.

[30] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. Megascale: Scaling large language model training to more than 10,000 gpus. *arXiv preprint arXiv:2402.15627*, 2024.

[31] Y-M Joo and Nick McKeown. Doubling memory bandwidth for network buffers. In *Proceedings. IEEE INFOCOM'98, the Conference on Computer Communications. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Gateway to the 21st Century (Cat. No. 98*, volume 2, pages 808–815. IEEE, 1998.

[32] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *Proc. NSDI*, 2019.

[33] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proc. SIGCOMM*, 2020.

[34] Luyang Li, Heng Pan, Xinchen Wan, Kai Lv, Zilong Wang, Qian Zhao, Feng Ning, Qingsong Ning, Shideng Zhang, Zhenyu Li, et al. Harmonia: A unified framework for heterogeneous fpga acceleration in the cloud. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 498–514, 2025.

[35] Qiang Li, Qiao Xiang, Derui Liu, Yuxin Wang, Haonan Qiu, Xiaoliang Wang, Jie Zhang, Ridi Wen, Haohao Song, Gexiao Tian, et al. From rdma to rdca: Toward high-speed last mile of data center networks using remote direct cache access. *arXiv preprint arXiv:2211.05975*, 2022.

[36] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: High precision congestion control. In *Proceedings of the ACM special interest group on data communication*, pages 44–58. 2019.

[37] Heng Liao, Bingyang Liu, Xianping Chen, Zhigang Guo, Chuanning Cheng, Jianbing Wang, Xiangyu Chen, Peng Dong, Rui Meng, Wenjie Liu, et al. Ub-mesh: a hierarchically localized nd-fullmesh datacenter network architecture. *arXiv preprint arXiv:2503.20377*, 2025.

[38] Xudong Liao, Han Tian, Chaoliang Zeng, Xinchen Wan, and Kai Chen. Astraea: Towards fair and efficient learning-based congestion control. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 99–114, 2024.

[39] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. Bgl:gpu-efficientgnn training by optimizing graph data i/o and preprocessing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 103–118, 2023.

[40] Yuan Liu, Wenxin Li, Yulong Li, Lide Suo, Xuan Gao, Xin Xie, Sheng Chen, Ziqi Fan, Wenyu Qu, and Guyue Liu. Fork: A dual congestion control loop for small and large flows in datacenters. In *Proceedings of the European Conference on Computer Systems 2025 Conference (EuroSys'25)*, pages 446–459, 2025.

[41] Jie Lu, Jiaqi Gao, Fei Feng, Zhiqiang He, Menglei Zheng, Kun Liu, Jun He, Binbin Liao, Suwei Xu, Ke Sun, et al. Alibaba stellar: A new generation rdma network for cloud ai. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 453–466, 2025.

[42] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-path transport for rdma in datacenters. In *15th USENIX symposium on networked systems design and implementation (NSDI 18)*, pages 357–371, 2018.

[43] Ralf Lübben, Philip Wette, and Sascha Gübner. Rttprobs: A rtt probing scheme for rtt aware multi-path scheduling. In *2021 IEEE 93rd Vehicular Technology Conference (VTC2021-Spring)*, pages 1–6. IEEE, 2021.

[44] Kai Lv, Jinyang Li, Pengyi Zhang, Heng Pan, Luyang Li, Shuihai Hu, Zhenyu Li, Gaogang Xie, Jingbin Zhou, and Kun Tan. Omnidma: Scalable rdma transport over wan. In *Proceedings of the 9th Asia-Pacific Workshop on Networking*, pages 135–141, 2025.

[45] Kai Lv, Heng Pan, Chengjun Jia, Jiaxing Zhang, Luyang Li, Jianer Zhou, Yanbiao Li, Zhenyu Li, and Gaogang Xie. Roundabout: Solving pfc deadlocks with distributed detection and buffer collaboration. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP)*, pages 1–12. IEEE, 2024.

[46] Yiqing Ma, Han Tian, Xudong Liao, Junxue Zhang, Weiyan Wang, Kai Chen, and Xin Jin. Multi-objective congestion control. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 218–235, 2022.

[47] Mellanox ConnectX-6 Product Brief. https://network.nvidia.com/sites/default/files/doc-2020/pb-connectx-6-en-card.pdf, 2020.

[48] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, et al. From luna to solar: the evolutions of the compute-to-storage networks in alibaba cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 753–766, 2022.

[49] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, et al. From luna to solar: the evolutions of the compute-to-storage networks in alibaba cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 753–766, 2022.

[50] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *Proc. SIGCOMM*, 2015.

[51] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

[52] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring endpoint congestion control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 30–43, 2018.

[53] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[54] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 327–341, 2018.

[55] Ns-3 Network Simulator. https://www.nsnam.org/, 2011.

[56] Vladimir Olteanu, Haggai Eran, Dragos Dumitrescu, Adrian Popa, Cristi Baciu, Mark Silberstein, Georgios Nikolaidis, Mark Handley, and Costin Raiciu. An edge-queued datagram service for all datacenter traffic. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 761–777, 2022.

[57] Konstantinos Prasopoulos, Ryan Kosta, Edouard Bugnion, and Marios Kogias. Sird: A sender-informed,receiver-driven datacenter transport protocol. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 451–471, 2025.

[58] Ringbuffer documentation. https://docs.majerle.eu/projects/lwrb/en/v1.2.0/index.html, 2019.

[59] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 123–137, 2015.

[60] Arjun Singhvi, Nandita Dukkipati, Prashant Chandra, Hassan MG Wassel, Naveen Kr Sharma, Anthony Rebello, Henry Schuh, Praveen Kumar, Behnam Montazeri, Neelesh Bansod, et al. Falcon: A reliable, low latency hardware transport. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 248–263, 2025.

[61] Parul Sohal, Michael Bechtel, Renato Mancuso, Heechul Yun, and Orran Krieger. A closer look at intel resource director technology (rdt). In *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, pages 127–139, 2022.

[62] Supplement to InfiniBand architecture specification volume 1 release 1.2.2 annex A17: RoCEv2 (IP routable RoCE). https://www.infinibandta.org/specs, 2014.

[63] Understanding Performance of PCI Express Systems. https://docs.amd.com/v/u/en-US/wp350, 2014.

[64] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.

[65] Midhul Vuppalapati, Saksham Agarwal, Henry Schuh, Baris Kasikci, Arvind Krishnamurthy, and Rachit Agarwal. Understanding the host network. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 581–594, 2024.

[66] Xinchen Wan, Luyang Li, Han Tian, Xudong Liao, Xinyang Huang, Chaoliang Zeng, Zilong Wang, Xinyu Yang, Ke Cheng, Qingsong Ning, et al. A generic and efficient communication framework for message-level in-network computing. In *IEEE INFOCOM 2025-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2025.

[67] Tao Wang, Xiangrui Yang, Gianni Antichi, Anirudh Sivaraman, and Aurojit Panda. Isolation mechanisms for high-speedpacket-processing pipelines. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1289–1305, 2022.

[68] Weitao Wang, Masoud Moshref, Yuliang Li, Gautam Kumar, TS Eugene Ng, Neal Cardwell, and Nandita Dukkipati. Poseidon: Efficient, robust, and practical datacenter cc via deployable int. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 255–274, 2023.

[69] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, et al. Srnic: A scalable architecture for rdmanics. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1–14, 2023.

[70] Zilong Wang, Xinchen Wan, Luyang Li, Yijun Sun, Peng Xie, Xin Wei, Qingsong Ning, Junxue Zhang, and Kai Chen. Fast, scalable, and accurate rate limiter for rdma nics. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages

568–580, 2024.
[71] Jeremiah J Wilke and Joseph P Kenny. Opportunities and limitations of quality-of-service (qos) in message passing (mpi) applications on adaptively routed dragonfly and fat tree networks. Technical report, Sandia National Lab.(SNL-CA), Livermore, CA (United States), 2020.
[72] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 459–471, 2013.
[73] Wenji Wu, Matt Crawford, and Mark Bowden. The performance analysis of linux networking–packet receiving. *Computer Communications*, 30(5):1044–1057, 2007.
[74] Francis Y Yan, Jestin Ma, Greg D Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 731–743, 2018.
[75] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, et al. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1345–1358, 2022.
[76] Guannan Zhang, Dinghuang Hu, and Dezun Dong. Rately: Accurate data center cc based on one-way delay. In *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 2759–2760. IEEE, 2023.
[77] Lingjun Zhu, Yifan Shen, Erci Xu, Bo Shi, Ting Fu, Shu Ma, Shuguang Chen, Zhongyu Wang, Haonan Wu, Xingyu Liao, et al. Deploying user-space tcp at cloud scale with luna. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 673–687, 2023.
[78] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proc. SIGCOMM*, 2015.

A Algorithm Deployment

The lifecycle of a CC algorithm integrated with Taurus comprises initialization, execution, result updates, and teardown. Below, we illustrate a typical deployment of a custom CC algorithm:

Initialization: Taurus exposes a unified algo_init API to standardize algorithm context and parameter initialization and to bind the algorithm to required events. Concretely, algorithms register their context and parameters via a user_algoname_setup interface, which accepts the algorithm-specific context and parameter structures. Event handlers are bound using algo->handle__event callbacks. The set of supported event types is summarized in Table 3.

Execution: The algorithm agent is compiled into a binary and deployed on the software side. During runtime, the agent consumes signals reported from the datapath and invokes the registered handlers (e.g., user_algoname_handle_*_event) in response to the corresponding events. Users implement these handler functions to realize their algorithm's decision logic.

Result updates: Algorithms publish outcomes via the result_update interface, which updates algorithm state, context, and related parameters and coordinates any necessary actions toward the datapath.

Teardown: Upon receiving a termination or interrupt signal, the agent performs cleanup by releasing algorithm contexts, freeing allocated resources, and unbinding event handlers.

```
#include "Taurus_algo.h"

// the unified entrance provided by the CC agent
ALGO_STATUS algo_init(struct pcc_algo *algo) {
    switch (algo->algo_type) {
        case user_algo1:
            algo->setup_algo = user_algo1_setup;
            algo->handle_ack_event = user_algo1_handle_ack_event;
            algo->handle_rtt_event = user_algo1_handle_rtt_event;
```

Table 3. Primary APIs and events provided to users. 1) *a*: algorithm; 2) *c*: context; 3) *p*: parameter; 4) *e*: event; 5) *r*: result; 6) *id*: flow index. *: defined by algorithms

Primary APIs	Description
*algo_init(*a)*	Initialize CC contexts and params.
*user_algo*_setup(*c,*p)*	Interface for CC setup.
*user_algo*_handle_*_event(*e, id, *c,*p)*	Interface for CC decisions.
*result_update(*c,*p,r,id)*	Issue update results to hardware.
*teardown(*a)*	Tear down a CC algorithm.
Events	**Description**
CC_ACK_EVT	Acknowledged packet event.
CC_ECN_EVT	ECN-marked packet event.
CC_OOO_EVT	Out-of-order packet event.
CC_RTO_EVT	Retransmission-timeout event.
CC_RTT_EVT	RTT probing result event.
CC_TX_EVT	Tx bytes & packets event.
*CC_CST_EVT**	Custom feedback events.
*CC_HOST_EVT**	Host congestion events.

```
            ...
        case user_algo2: ...
        default: return DEFAULT_ALGO;
    }
}

// users define their own setup logic
uint32_t user_algo1_setup(struct user1_algo_conext &ctx, struct user1_algo_param &param) {
    init_algo1_context(ctx);
    init_algo1_param(param);
}

// user's CC strategies that react to ack events
uint32_t user_algo1_handle_ack_event(struct ack_event &acks, uint32_t &id,
    struct user1_algo_conext &ctx, struct user1_algo_param &param) {
    do congestion control algorithm for ack event;
    result_update(ctx, param, result, id);
}

// user's CC strategies that react to rtt events
uint32_t user_algo1_handle_rtt_event(struct ack_event &acks, uint32_t &id,
    struct user1_algo_conext &ctx, struct user1_algo_param &param) {
    do congestion control algorithm for rtt event;
    result_update(ctx, param, result, id);
}

// tear down a CC algorithm
uint32_t teardown(struct pcc_algo *algo) {
    do tear down using the interrupt interface;
}
```

B Experiment Artifact

B.1 Artifact check-list

The following lists the experiment environments, compilation and testing tools, and project repository used in our experiments. We provide the exact version information and installation links.

- **Program**: C/C++
- **Compilation**: g++ 11.1.0/5.4.0, gcc-11.0.0/5.4.0
- **Kernel**: Linux 4.18
- **MLC**: https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html
- **CPU sampling**: https://man7.org/linux/man-pages/man1/top.1.html
- **iperf3**: iperf-3.16 https://github.com/esnet/iperf/releases
- **perftest**: perftest-24.01.0 https://github.com/linux-rdma/perftest/releases/
- **netperf**: netperf-2.6.0 https://github.com/HewlettPackard/netperf
- **PCIe Model**: https://dl.acm.org/doi/pdf/10.1145/3230543.3230560
- **NS3 Simulation Repo(HPCC)**: https://github.com/alibaba-edu/High-Precision-Congestion-Control/tree/master
- **NS3 Simulation Repo(Swift, Bolt)**: https://github.com/serhatarslan-hub/bolt_cc_ns3
- **CCP Repo**: https://github.com/ccp-project/ccp-kernel.git
- **DOCA-PCC Repo**: https://developer.nvidia.com/doca-downloads
- **HostCC Repo**: https://github.com/Terabit-Ethernet/hostCC

B.2 Installation & Launch

Use the following guides to install each framework and algorithm in the testbed and simulation environments.

```
# for CCP in the testbed
do the installation by following guides at
↪  https://ccp-project.github.io/ccp-guide/setup/index.html
do the launch by following guides at
↪  https://ccp-project.github.io/ccp-guide/running.html

# for DOCA-PCC in the testbed
do the installation by following guides at     https://docs.nvidia.com/doca/archiv⌋
↪  e/2-5-2/nvidia+doca+installation+guide+for+linux/index.html
do the launch by following guides at        https://docs.nvidia.com/doca/archive/2-⌋
↪  5-2/nvidia+doca+pcc+application+guide/index.html

# for HostCC in the testbed
do the installation by following guides at
↪  https://github.com/Terabit-Ethernet/hostCC/tree/main/utils
do the launch by following guides at
↪  https://github.com/Terabit-Ethernet/hostCC/tree/main/scripts

# for CC algorithms in NS3-Simulation
do the installation by following guides at
↪  https://github.com/alibaba-edu/High-Precision-Congestion-Control/
do the launch by following guides at
↪  https://github.com/alibaba-edu/High-Precision-Congestion-Control/
```

B.3 Experiment workflow & result collection.

In the following, we detail the concrete experiment workflows as well as the methods used to collect and analyze results in each evaluation part.

B.3.1 Micro-benchmarks. We deploy Taurus on the testbed and initialize the Taurus agent. To evaluate the system overhead, multiple flows are generated (ranging from 1 to 16) from multiple sender hosts to a single receiver using `perftest`, and throughput is measured once the system reaches steady state. The CPU utilization of the Taurus agent is sampled periodically using Linux `top` to characterize software overhead. In parallel, FPGA hardware registers record both message reporting and result update times. The software periodically reads these registers, computes differences between consecutive readings, and thereby quantifies the message rates.

B.3.2 Algorithm performance. Taurus is evaluated in both testbed and simulation environments to measure throughput, latency, queue occupancy, and FCT. In the testbed, `perftest` is used to evaluate throughput under diverse scenarios as described in §5.3, while `netperf` measures latency and FCT. In simulation, algorithms execute within `NS3` with a PCIe latency model for comparisons with alternative CC algorithms and in-datapath implementations. `NS3` provides detailed metrics on throughput, latency, queue occupancy, and FCT.

B.3.3 Framework comparison. Taurus, `DOCA-PCC`, and `CCP` are deployed on the testbed to compare framework performance. Local host congestion is induced using `MLC`, with traffic volume configurable via MLC parameters to emulate CPU-to-memory bottlenecks. Fabric congestion is generated by initiating multiple sender flows targeting the same receiver. Throughput measurements are collected using `perftest` and `iperf3`, while `netperf` captures the latency. This setup enables Taurus to be evaluated under different congestion scenarios.

Received June 2025; accepted September 2025