CEIO: A Cache-Efficient Network I/O Architecture for NIC-CPU Data Paths

Bowen Liu*
Hong Kong University of Science and Technology
Hong Kong, China
liubw@ust.hk

Xinyang Huang*
Hong Kong University of Science and Technology
Hong Kong, China
xhuangci@connect.ust.hk

Qijing Li
Hong Kong University of Science and Technology
Hong Kong, China
qlicw@connect.ust.hk

Zhuobin Huang
University of Electronic Science and Technology of China
Chengdu, China
zobin1999@std.uestc.edu.cn

Yijun Sun
Hong Kong University of Science and Technology
Hong Kong, China
ysuneb@connect.ust.hk

Wenxue Li
Hong Kong University of Science and Technology
Hong Kong, China
wlicv@connect.ust.hk

Junxue Zhang
University of Science and Technology of China
Hefei, China
snowzjx@ustc.edu.cn

Ping Yin
Inspur Cloud
Jinan, China
yinping@inspur.com

Kai Chen
Hong Kong University of Science and Technology
Hong Kong, China
kaichen@cse.ust.hk

Abstract

Efficient Input/Output (I/O) data path between NICs and CPUs/-DRAMs is critical for supporting datacenter applications with high-performance network transmission, especially as link speed scales to 100Gbps and beyond. Traditional I/O acceleration strategies, such as Data Direct I/O (DDIO) and Remote Direct Memory Access (RDMA), perform suboptimally due to the inefficient utilization of the Last-Level Cache (LLC). This paper presents `CEIO`, a novel cache-efficient network I/O architecture that employs proactive rate control and elastic buffering to achieve zero LLC misses in the I/O data path while ensuring the effectiveness of DDIO and RDMA under various network conditions. We have implemented `CEIO` on commodity SmartNICs and incorporated it into widely-used DPDK and RDMA libraries. Experiments with well-optimized RPC framework and distributed file system under realistic workloads demonstrate that `CEIO` achieves up to 2.9× higher throughput and 1.9× lower P99.9 latency over prior work.

CCS Concepts

• **Networks** → **Data center networks**; **Network design principles**; • **Software and its engineering** → **Operating systems**.

*Bowen Liu and Xinyang Huang contributed equally to this research.

SIGCOMM '25, September 8–11, 2025, Coimbra, Portugal

ACM ISBN 979-8-4007-1524-2/25/09
https://doi.org/10.1145/3718958.3750488

Keywords

Host Network, Network I/O, Cache Efficiency

ACM Reference Format:
Bowen Liu, Xinyang Huang, Qijing Li, Zhuobin Huang, Yijun Sun, Wenxue Li, Junxue Zhang, Ping Yin, and Kai Chen. 2025. CEIO: A Cache-Efficient Network I/O Architecture for NIC-CPU Data Paths. In *ACM SIGCOMM 2025 Conference (SIGCOMM '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3718958.3750488

1 Introduction

The network I/O system is designed to facilitate data transfer among host I/O components (*e.g.*, NIC, LLC, DRAM, and CPU), serving as the network entrance for host applications. A typical I/O process begins with the NIC receiving a packet from the network, after which the NIC firmware copies the packet to an I/O buffer in the host memory. Traditional network I/O architectures predominantly rely on host CPUs to handle the remaining I/O operations, such as transport protocol processing and payload processing. However, as network bandwidth scales to 100/200/400Gbps, CPUs become a crucial bottleneck in the I/O data path—the processing speed of CPUs lags significantly behind the network transmission rate—failing to achieve the expected line-rate throughput and µs-scale latency [9, 10, 18, 19, 35, 65, 84].

To address the issue, hardware vendors such as Intel, AMD, and NVIDIA have introduced techniques like DDIO [8, 15] and RDMA [39, 77]. These techniques aim to accelerate the I/O process by enabling NICs to issue direct memory access (DMA) requests to the LLC and offloading critical I/O operations to the NICs respectively. Although promising, our community has noticed that the inefficient LLC utilization leads to suboptimal I/O performance [20, 69, 70, 73, 76]. Specifically, LLC misses occur when the volume of in-flight data within the I/O system exceeds the LLC capacity, forcing additional data retrieval from DRAM.

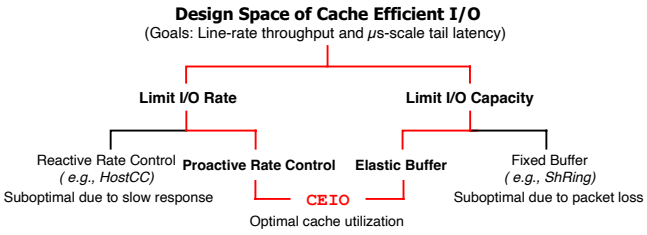


Figure 1: Design space of optimizing LLC efficiency in CPU-NIC I/O data paths.

Under such conditions, the benefits of DDIO and RDMA are significantly diminished due to increased CPU cycle consumption and memory bandwidth usage (§2.2). Consider a 200Gbps link transmitting 1024B packets, each I/O operation have to complete within only 41.8 nanoseconds to achieve the line-rate. If an LLC miss occurs during packet processing, a delay over 100ns will happen and the desired performance cannot be promised [17, 20]. Even worse, multi-core parallelization suffers from low efficiency due to intensified cache contention and poor scalability of concurrent DRAM accesses, leading to substantial CPU cycle wastage and thus unacceptable for datacenter applications [65, 73].

To unleash the potential of DDIO and RDMA, recent efforts have explored two directions to mitigate LLC misses for I/O efficiency (Figure 1): limiting the I/O upload rate and limiting the I/O accessible capacity. The first direction, represented by Host Congestion Control (HostCC) [1] and its RDMA variant RHCC [74], reactively reduces the NIC's DMA rate upon detecting I/O congestion signals to mitigate LLC contention. The second direction, represented by ShRing [61], limits the I/O accessible capacity by aggregating all I/O buffers into a shared ring, fixing its size smaller than the LLC capacity to eliminate LLC misses.

However, as revealed in §2.3, directly applying these two approaches suffers from *slow response* and *packet loss* in practice: (1) the reactive rate control, such as HostCC, is triggered by LLC misses because it relies on LLC congestion signals, resulting in slow I/O processing in CPUs; (2) the fixed buffer, such as ShRing, requires to frequently trigger network congestion control algorithms (CCAs) to prevent packet loss, resulting in a slow and unstable network ingress rate. In our experiments, we observe these limitations cause up to 1.9× performance degradation in realistic network environments, such as dynamic flow distribution and network burst (Figure 4). This motivates us to explore a new I/O architecture that features proactive I/O rate control and elastic buffering (Figure 1), thereby enabling optimal LLC utilization and stable performance under various network conditions.

Guided by this insight, we propose `CEIO` (§3), an I/O architecture that achieves optimal LLC utilization by developing an I/O manager in the NIC—the entrance of I/O data path. At its core, we proactively regulate the producer's and consumer's rates of I/O buffers between the NIC and the CPU/DRAM via credit-based flow control (§4.1). Specifically, all network flows need to apply for credits from the `CEIO` flow controller before uploading, where the total credits correspond to the LLC capacity. This ensures that LLC will not be overflowed. On the other hand, `CEIO` elastically buffers exceeded I/O data in the on-NIC memory to prevent packet loss (§4.2), featuring order-preserving and asynchronous access mechanisms that enhance end-to-end I/O efficiency.

We have fully implemented `CEIO` on NVIDIA BlueField platform [62] and incorporated it with two popular I/O frameworks, DPDK [22] and RDMA [23] (§5). In §6, we evaluate `CEIO` with well-optimized NIC-CPU data path implementations across realistic workloads—a key-value store built on eRPC [37] and an in-memory DFS named LineFS [36]—to understand benefits of `CEIO`'s core idea. In a 200Gbps network configuration, `CEIO` achieves 1.2× to 2.5× improvements in terms of throughput and 1.4× to 1.9× improvements in terms of P99 and P99.9 latency compared to previous I/O optimizations such as HostCC and ShRing. Additionally, `CEIO` maintains effective and stable performance under various network conditions, such as dynamic flow distribution and network burst.

`CEIO` is open-sourced at https://github.com/axio-project/ceio. This work does not raise any ethical issues.

2 Background and Motivation

2.1 Network I/O Datapath

The network Input/Output (I/O) subsystem facilitates data transfer between hosts (e.g., CPU processors and DRAM) and network devices (e.g., NICs). Its primary goal is to deliver line-rate throughput and µs-scale latency, providing efficient network services for modern distributed systems [16, 25, 56, 57, 63, 67]. A legacy I/O system comprises four key hardware components that collaboratively complete the data movement: NIC firmware, PCIe Direct Memory Access (DMA) engine, host memory controller, and CPU processor. As illustrated in Figure 2, when a packet arrives at the NIC, it undergoes the following stages [21, 60]:

1. The firmware retrieves the address of an available buffer in the host memory (encapsulated in a descriptor) and issues a DMA request to the PCIe DMA engine.
2. The DMA engine encapsulates the packet into Transaction Layer Packets (TLPs), transfers it across the PCIe interconnect, and writes it to the Integrated I/O (IIO) buffer.
3. The host memory controller retrieves the packet from IIO, writes the packet to DRAM, and completes this DMA request.
4. The firmware notifies the CPU of the arrival of a new packet using either an interrupt or a polling mechanism.
5. Finally, the CPU handles the remaining I/O operations (*e.g.*, network stack processing) through standard memory access.

Modern I/O systems primarily utilize two techniques to accelerate the above data movement: Direct Data I/O (DDIO) [15, 29, 68] and Remote Direct Memory Access (RDMA) [26]. Different datacenter applications benefit from these two techniques in distinct ways, as shown in Figure 3.

1 **NIC → LLC → CPU:** DDIO accelerates 3 by allowing the memory controller to directly write received data to the Last-Level Cache (LLC) [41, 76]. This significantly reduces memory access latency of CPUs, providing a substantial performance boost for applications that rely on real-time CPU processing, such as Remote Procedure Call (RPC) requests, network function processing, and database operations.

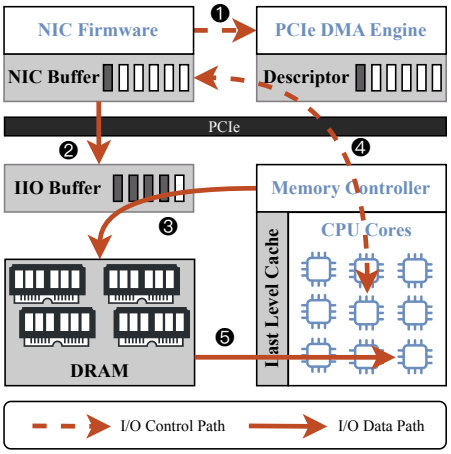


Figure 2: Illustration of the I/O data path between NIC and CPU/DRAM, without DDIO or RDMA acceleration (details are described in §2.1).

(2) **NIC → LLC → DRAM:** RDMA bypasses CPU-involved I/O operations (*i.e.*, (4) and (5)) by offloading network stack processing to the NIC hardware. This allows the data movement to be completed without CPU involvement, which is typically considered capable of achieving line-rate transmission [38, 40, 45, 79, 80]. This technique is particularly suitable for applications that require minimal CPU involvement, such as large file transfers in distributed file systems (DFS) and data exchanges in AI-related scenarios[1].

Finally, the accelerated data flows in I/O systems (I/O flows) can be categorized into two types. In the following sections, we refer to (1) as `CPU-involved` flows and (2) as `CPU-bypass` flows.

2.2 Managing LLC is Necessary

To fully leverage both `CPU-involved` and `CPU-bypass` flows for accelerating data movement, effective management of the LLC is crucial [19, 84]. Once DDIO is enabled, all I/O data will first be transmitted to the LLC, whether from `CPU-involved` or `CPU-bypass` flows. However, it is important to note that the LLC is limited (*i.e.*, typically only dozens of megabytes in total) and shared (*i.e.*, among all CPU cores and NICs within a NUMA node). As a result, LLC misses easily occurs, we take two common scenarios as examples:

- Excessive in-flight data volume: When the volume of in-flight I/O data significantly exceeds the limited LLC capacity[2], LLC misses can easily occur. This situation commonly arises under heavy network load, such as during sustained line-rate transmission or sudden bursts. For example, when running an RPC system (consists of `CPU-involved` flows), frequent cache misses are observed when incoming packets arrive faster than the CPU can process them. In such cases, subsequent packets overwrite earlier ones in the LLC, prematurely evicting them to DRAM before the CPU accesses them.
- Coexistence of `CPU-involved`/`CPU-bypass` flows: When heterogeneous I/O flows coexist, LLC misses are frequent due to the shared nature. For instance, deploying an RPC system (consists of `CPU-involved` flows) alongside a distributed file system (consists of `CPU-bypass` flows) in the same server, which is one common setup in public clouds such as Amazon EC2 multitenant servers. In this case, the `CPU-bypass` flow would continuously flush the LLC, evicting the `CPU-involved` flows' packets to DRAM before the CPU processes them and thus causing LLC misses.

Both `CPU-involved` and `CPU-bypass` flows suffer from suboptimal when LLC utilization is inefficiency, resulting up to 1.5× performance degradation according to recent works [1, 20, 61, 73, 74]. We show two concrete examples to explain the performance degradation, as illustrated in Figure 3:

• *Impact on CPU-involved flows* (1): Once the data is evicted from the LLC to DRAM before the CPU processes it, the CPU must spend additional cycles to retrieve the data from DRAM. Then, the I/O data paths are extented to (3) **NIC → LLC → DRAM → LLC → CPU**, which directly increases the latency of each I/O operation. Furthermore, the extended I/O flows have two negative impacts on the throughput metric [20]: (1) CPU cycles getting burden due to the additional memory access overhead, failing to achieve expected packet processing rates; and (2) the extended latency prevents commodity hardware from processing packets at line rate, where the access latency of DRAM is typically greater than the interarrival time between two packets (*e.g.*, 41.8 nanoseconds for 1024B packets in 200Gbps network settings). Thus, both latency and throughput metrics degrade, leading to suboptimal I/O performance.

• *Impact on CPU-bypass flows* (2): The process of the CPU retrieving missed packets from DRAM would block `CPU-bypass` flows, significantly degrading their throughput [73]. The underlying reasons are twofold: (1) (1) (issue DMA request) requires available PCIe credits, which can be exhausted due to the long processing latency of `CPU-involved` flows; and (2) the throughput of (3) (memory controller fetches the packet from IIO to DRAM) degrades due to the additional memory access overhead, occupying the memory bandwidth that required by `CPU-bypass` flows. Consequently, `CPU-bypass` flows degrade to (4) **NIC → (block) LLC → (slow) DRAM**, which is less efficient than the original (2) one.

Summary: When LLC utilization is inefficient, achieving line-rate and µs-scale latency I/O becomes nearly impossible due to increased consumption on CPU cycles and memory bandwidth. Management is necessary to mitigate the LLC contention, unleash the potential of both `CPU-involved` and `CPU-bypass` flows, and prevent performance degradation.

2.3 Existing Works and Limitations

As the representative of controlling the I/O rate, HostCC reduces the NIC's DMA rate[3] upon detecting I/O congestion signals (*i.e.*, IIO buffer occupancy) that indicates the I/O flow cannot be consumed by the CPU or memory controller in its input rate. For limiting the I/O capacity, ShRing [61], constrains the I/O-accessible buffers

[1]For GPUs lacking NVIDIA GPUDirect [50] and PCIe Peer-to-Peer (P2P) capabilities, such as the NVIDIA Geforce Series [48, 49], all I/O operations must involve DRAM, resulting in a data flow pattern similar to (2).
[2]The capacity depends on the DDIO configuration [20], typically two cache ways of LLC.
[3]HostCC achieves this by triggering network congestion control algorithms such as DCTCP.

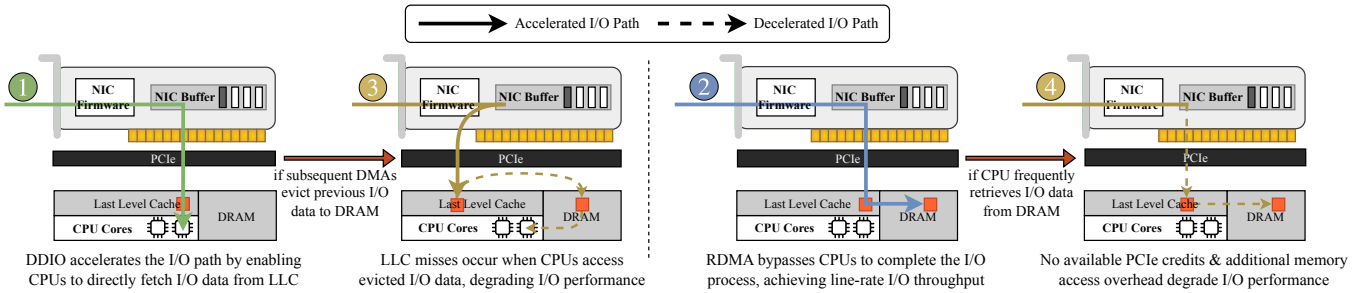
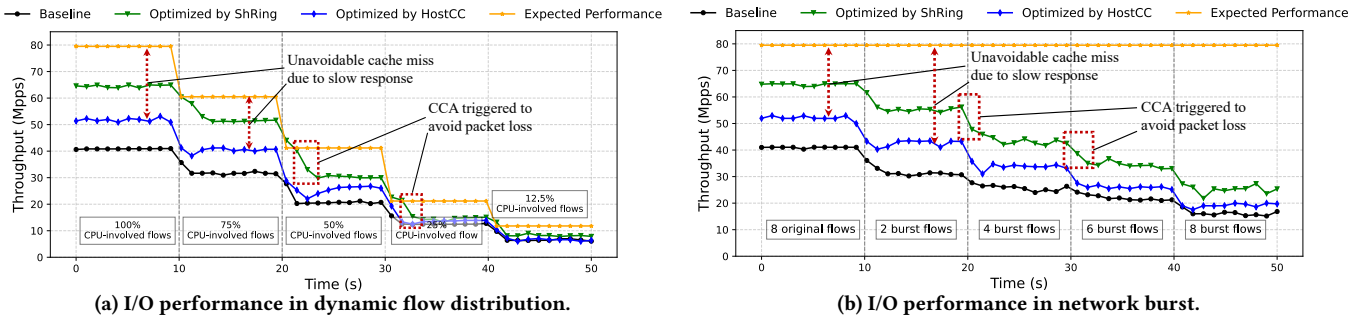


Figure 3: Four types of I/O data paths. From left to right are `CPU-involved` flows with DDIO acceleration, `CPU-involved` flows with inefficient LLC, `CPU-bypass` flows with RDMA acceleration, and `CPU-bypass` flows with inefficient LLC.



(a) I/O performance in dynamic flow distribution. **(b) I/O performance in network burst.**

Figure 4: The fundamental limitations of existing methods such as ShRing and HostCC lead to poor I/O performance, particularly due to the slow response and packet loss. Expected performance is calculated using the number of `CPU-involved` flows and the single core throughput of ShRing with sufficient LLC.

smaller than the LLC capacity, ensuring all in-flight data remains within the LLC and eliminating LLC misses.

While promising, we find they face two fundamental limitations: (1) reactive control suffers from slow response, and (2) fixed buffer risks packet loss, both of which result in suboptimal performance in practice. To demonstrate the impact of these limitations, we conduct experiments under two common datacenter scenarios: dynamic flow distribution and network burst.

Experiment Setup: We conduct experiments on two servers configured for 200Gbps network. Each server is equipped with two Intel Xeon Silver 4309Y CPU (16 cores per CPU, 2.8GHz for base frequency and 3.6GHz for turbo frequency), one NVIDIA BlueField 3 SmartNIC with PCIe 5.0×16 interconnection, and 512GB DDR4 3200MT/s DIMMs connected to 8 memory channels. The servers are configured to handle 200Gbps network traffic, constrained by their link capacities.

We deploy two I/O data paths for `CPU-involved` and `CPU-bypass` flows: a key-value store based on eRPC [37], built upon DPDK, and an in-memory DFS named LineFS [36], built upon RDMA (details see §6.1). DCTCP [2] is adopted for basic network rate control. To ensure sufficient computational capacity and prevent performance degradation caused by context switches, we dedicate one CPU core to each I/O flow. Million packets per second (Mpps) is used as the throughput metric for the I/O systems.

Degradation on Dynamic Flow Distribution: We first evaluate the performance of the two kinds of I/O flows in a dynamic flow distribution scenario. Initially, eRPC is deployed to handle eight `CPU-involved` flows, and every 10 seconds, two of these flows are replaced with `CPU-bypass` flows handled by LineFS.

The throughput of `CPU-involved` flows, shown in Figure 4a, reveals the following observations: (a) HostCC and ShRing improve the throughput by up to 1.3× and 1.7×, respectively, demonstrating the effectiveness of LLC optimizations. (b) However, when the flow distribution changes, the throughput improvement falls significantly lower than expectations. For HostCC, its slow response leads to unavoidable cache misses (about 70%) when generating congestion signals for DCTCP. These cache misses prevent HostCC from adjusting the DMA rate at the optimal timing, resulting in performance degradation of up to 1.9× compared to the expected performance (calculated with infinite LLC). For ShRing, its fixed I/O buffer configuration frequently triggers DCTCP to limit the network transmission rate. This occurs because the newly arrived `CPU-bypass` flows consume a portion of the I/O buffers allocated to `CPU-involved` flows. As a result, `CPU-involved` flows are forced to reduce their sending rates by up to 1.6× to avoid packet drops (realized by DCTCP).

Degradation on Network Burst: Next, we evaluate the performance of `CPU-involved` flows in a network burst scenario. We barely deploy eRPC to handle eight `CPU-involved` flows, with two additional burst `CPU-involved` flows (handled by two extra cores) arriving every 10 seconds. The results in Figure 4b indicate that the performance degradation is even more pronounced compared to the dynamic flow distribution scenario. This is because the increase in the total number of flows intensifies LLC contention, further amplifying the limitations of slow response and fixed I/O buffer.

	Limitations	I/O Performance
Reactive Rate Control (represented as HostCC)	The slow response limitation incurs unavoidable LLC misses	Slow I/O path processing rate
Fixed Buffer (represented as ShRing)	Unexpectedly trigger CCA to avoid packet loss	Slow network transmission rate
CEIO	Introduce **proactive rate control** and **elastic buffer** to address previous limitations	**Near line-rate** performance

Table 1: Comparison of different I/O optimization approaches

Our Insight: The slow response in reactive rate control (*e.g.*, HostCC) leads to unavoidable LLC misses, resulting in slow I/O path processing. Meanwhile, the packet loss in fixed buffering (*e.g.*, ShRing) leads to frequent triggers of CCAs, resulting in slow network transmission rate. Both factors contribute to suboptimal performance. These findings motivate us to incorporate proactive rate control to intervene I/O rates, along with elastic buffering to store exceeded I/O data, **at the entrance of the I/O system**—the NIC—before packets are DMAed to the LLC.

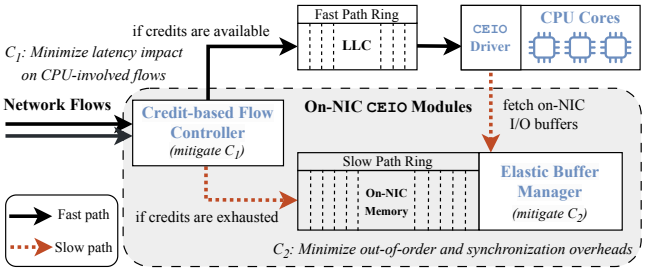


Figure 5: The overview architecture of CEIO.

3 CEIO Overview

In this paper, we propose a cache efficient I/O architecture, named CEIO, aiming to optimize LLC efficiency by addressing the fundamental limitations of existing solutions (Table 1). The high-level workflow of CEIO is illustrated in Figure 5.

NIC-driven, credit-based flow control. When a packet arrives at the NIC, it consumes a credit before initiating a DMA request. The total number of credits corresponds to the LLC capacity, with credits being consumed by newly arrived packets and replenished upon they have been processed. If a credit is successfully obtained, the packet proceeds through the legacy I/O process (Figure 2). Otherwise, the packet will be buffered in the on-NIC memory (*e.g.*, 16GB in commercial platform BlueField 3). Intuitively, the flow control proactively limit I/O rate before the LLC misses occur, so that prevent (1) degrades to (3) and (2) degrades to (4). A credit-based flow controller (§4.1), implemented on the NIC, is responsible for managing this process.

Elastic on-NIC & host I/O buffer management. The elastic on-NIC buffer stores excessive packets rather than drops them, effectively avoiding unexpected CCA triggers that could degrade network performance. CPUs access the on-NIC buffer to fetch packets via ring buffers and DMA, similar to the legacy I/O process. Specifically, when a CPU thread calls the `recv()` API (§5), CEIO driver will poll the legacy I/O ring and the on-NIC buffer ring registers to check if there are packets arrived. If the on-NIC buffer ring is not empty, the driver will initiate a DMA read request to fetch the packets to the host memory. Then, the `recv()` API returns the number of packets that have been stored in the host memory to the application for further processing. To support this functionality, an elastic buffer manager (§4.2) is required.

The proposed I/O architecture can be divided into a fast path and a slow path. The fast path, defined as *NIC → LLC → CPU/DRAM*, achieves line-rate throughput and µs-scale latency easily due to direct access to the LLC without incurring misses. However, the slow path, defined as *NIC → on-NIC Memory → CPU/DRAM*, introduces additional overheads, leading to two key technical challenges:

• ***How to address high latency caused by slow path*:** The slow path incurs higher latency, as the CPU must fetch data from the on-NIC buffer rather than directly from the LLC. This overhead, introduced by data traversal over the PCIe interconnect between the NIC and CPU, can reach up to 1000ns [12, 28, 47]. While this latency is inherent to the architecture, it poses a particular challenge for `CPU-involved` flows, where the CPU stalls while waiting for incoming packets. To address this, CEIO prioritizes keeping `CPU-involved` flows on the fast path. A common strategy is to assign higher priority to these flows. Though straightforward, allowing users to tag flow priorities introduces additional complexity for application developers and raises potential fairness and security concerns [11]. In this paper, we address this challenge by designing a credit allocation and release approach within the credit-based flow controller (§4.1). This flow control ensures that `CPU-involved` flows are more likely to utilize the fast path, achieving higher priority based solely on network information, such as message size, network throughput, flow completion time, etc.

• ***How to minimize the interaction overheads in slow path*:** Slow path introduces two extra interaction overheads: (1) Some packets may be buffered on the CPU side while others remain on the NIC side, resulting in a need of reordering mechanism, which typically consumes a substantial amount of CPU cycles. (2) Synchronous packet access across the PCIe interconnection (from CPU to on-NIC buffer) incurs long CPU pipeline stalls and degrades the throughput of the slow path [47]. These overheads make it challenging to achieve line-rate performance for both `CPU-involved` and `CPU-bypass` flows. In this paper, we tackle this challenge by designing the elastic buffer manager (§4.2) that avoids out-of-order packets and reduces synchronous access overhead.

4 CEIO Design

This section presents the detailed design of CEIO. We begin by introducing the credit-based flow controller for improving the fast path utilization (§4.1) and subsequently describe the elastic buffer manager for optimizing the slow path efficiency (§4.2). To enhance clarity, we outline our design choices for the flow controller and

buffer manager, focusing on how they address the challenges highlighted in §3, alongside a detailed explanation of the workflows for both the fast and slow paths.

4.1 Proactive, Credit-based Flow Control

Multiple Priority Queues V.S. Lazy Credit Release. The primary design goal of the credit-based flow controller is to ensure that `CPU-involved` flows are more likely to access the fast path. One potential approach is to adopt flow priority scheduling, such as Multiple Priority Queues (MPQs) [6, 42], and we attempt to tag `CPU-involved` flows with higher priority. However, we find that the fundamental principle of network flow scheduling—granting higher priority to shorter flows—is not well-suited for our system.

Consider PIAS [6] as an example, where the priority of a flow is determined by its active time. Specifically, all flows experience priority decay over time, transitioning from high to low priority. Based on the assumption that datacenter network flows tend to follow a long-tail distribution [3, 6]—*i.e.*, most flows are short, while a small percentage are very large—this mechanism ensures that short flows are completed in high-priority queues, while long flows are degraded to low-priority queues due to the priority decay. However, directly applying MPQs to an I/O system does not guarantee that `CPU-involved` flows will access the fast path as expected, since `CPU-involved` flows are not always short (*e.g.*, continuous RPC requests, video streaming, or flows in overlay networks). This raises a question: can we design a solution similar to MPQs that lets `CPU-bypass` flows are more likely to degrade to lower priority (*i.e.*, the slow path) than `CPU-involved` flows?

In this paper, we achieve a similar effect to MPQs by designing a lazy credit release mechanism, where credits are released only after a batch of messages is processed. This design choice is guided by the following observations: (a) In high-speed networks, `CPU-involved` flows are efficiently consumed by CPU cores using a polling method [22], meaning packets do not accumulate in I/O buffers as long as the CPU's processing rate exceeds the network rate; (b) The message size of `CPU-bypass` flows is typically very large [5, 34, 43, 64, 75, 83], and applying the lazy release results in slow credit replenishment. This, in turn, quickly uses up credits assigned to this flow. A concrete example is the NCCL RDMA library [56], where an RDMA Write-with-immediate is issued after a batch of RDMA write operations to signal the receiver that the network transmission is complete. In such cases, `CEIO` driver is called after receiving the RDMA Write-with-immediate message (via interrupt or polling) and the credits are replenished. This means most of RDMA write operations are processed in the slow path since the credits for this flow are exhausted very soon. As a result, we have the opportunity to reallocate credits from flows in the slow path (more likely to be `CPU-bypass` flows) to flows in the fast path (more likely to be `CPU-involved` flows).

Based on the above design choice, we propose the basic workflow of the credit-based flow controller, as illustrated in Figure 6. Upon establishing a connection, the flow controller allocates credits and offloads a steering rule to the low-level flow engine [14, 53, 81]—a reconfigurable match-action engine (RMT engine) to redirect received packets to either the fast or slow path. This steering rule initially directs all flows' packets to the fast path via DMA. The

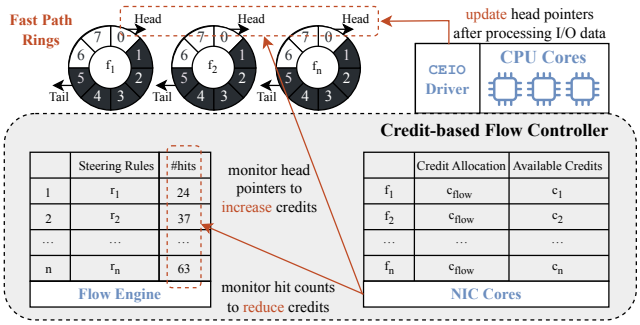


Figure 6: The basic workflow of credit-based flow controller.

flow controller manages credits for all flows and continuously polls counters in the steering flow table to track credit consumption. When a flow exhausts its credits, the flow controller updates the steering rule to DMA packets into on-NIC memory, transferring packet ownership to the elastic buffer manager (§4.2). Credits are replenished by the `CEIO` driver, only after the CPU or memory controller has processed a batch of message.

In the rest of this section, we present the detailed design of the credit-based flow controller and address the following questions:

Q1 How should credits be allocated when new flows arrive?

Q2 How to handle the flows in slow path, preventing them from flushing the cache and giving chance to upgrade to fast path?

Q3 How to reallocate credits and scale to large number of flows?

Detailed Design of Credit-based Flow Control. Let the set of network flows be denoted as $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$, the set of each flow's credits as $C = \{c_1, c_2, \ldots, c_n\}$, and the set of steering rules as $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$. Each flow is mapped to a specific NIC queue. To ensure LLC efficiency, the total number of credits, C_{total}, is determined by the available LLC space:

$$C_{total} = \frac{Size_{LLC}}{Size_{buf}} \tag{1}$$

$Size_{buf}$ represents the management granularity of LLC (including allocation, consumption, and replenishment), which is 64B in practice for an x86-based CPU architecture to maximize the LLC utilization. For simplicity, we suppose the $Size_{buf}$ is I/O buffer size, which stores a network packet. Then, in our testbed environment (Intel Xeon Silver 4309Y [32]), the available LLC size is configured to 6MB (using 6 out of 12 cache ways for DDIO), and the I/O buffer size is set to 2KB (considering the MTU is configured to 1500B). This results in a C_{total} value of 3000. According to our design, the initial credit allocation for one of n flows is given by:

$$C_{flow} = \frac{C_{total}}{n} \tag{2}$$

As previously mentioned, $\mathcal{R}$ is stored in the RMT engine, while credits are recorded and managed by the flow controller, implemented on on-NIC cores such as the ARM cores in NVIDIA BlueField DPUs. Credit allocation begins upon connection establishment. For instance, when a flow f_1 is established, the flow controller allocates $c_1 = 3000$ credits to f_1 and creates r_1 to enable legacy I/O for the flow. Subsequently, each packet of f_1 is directly DMAed to host memory and notifies the CPU through either an interrupt or a

Algorithm 1 CEIO Credit Management Strategy.

Input:
n: Current flow number.
m: New arrived flow number.
C_{total}: The total credit number of Rx host
1: // ***Credit assignment process.***
2: $C_{flow} = \frac{C_{total}}{n+m}$
3: **for** $i = 1$ to n **do**
4: **if** $c_i \geq \frac{m}{n} C_{flow}$ **then**
5: **for** $j = n$ to $n+m$ **do**
6: $c_j \leftarrow c_j + \frac{1}{n} C_{flow}$; $c_i \leftarrow c_i - \frac{1}{n} C_{flow}$
7: **else**
8: insert c_i into I
9: // *I is the set of flows lacking sufficient credits to allocate.*
10: **for** $j = n$ to $n+m$ **do**
11: // *Allocate credits of c_i uniformly to remaining flows.*
12: $c_j \leftarrow c_j + \frac{1}{m} c_i$
13: $o_j^i \leftarrow \frac{1}{n} C_{flow} - \frac{1}{m} c_i$ // *o_j^i is the credits c_i owes to c_j*
14: $c_i \leftarrow c_i - \frac{1}{m} c_i$
15:
16: // ***Credit release process,*** *γ_i is the released credits of f_i.*
17: **for** $f_i \notin I$ **do**
18: $c_i = c_i + \gamma_i$
19: **for** $f_i \in I$ **do**
20: **for** $f_j \in \{f_j | o_j^i > 0\}$ **do**
21: $o_j^i = o_j^i - max(o_j^i, \frac{\gamma_i}{m})$
22: $\gamma_i = \gamma_i - max(o_j^i, \frac{\gamma_i}{m})$
23: $c_i = c_i + \gamma_i$
24: **if** $o^i = 0$ **then**
25: Remove f_i from I.
26: **return**

polling mechanism (Figure 2). An on-NIC core is assigned to monitor this flow, tracking the hit count of r_1 and the head pointer of f_1's legacy ring buffer to adjust the corresponding credits c_1 as necessary. When c_1 is exhausted, the flow controller updates the action field of r_1 to DMA packets into on-NIC memory. A detailed discussion of lazy credit release is deferred to §4.2, as it is closely tied to ring buffer design, including operations like pointer updates.

Next, we address the previously raised questions to elaborate on additional details of the credit-based flow controller.

• **Q1: Handling new flows.** The flow controller employs the credit management strategy outlined in Algorithm 1 to reallocate C for new flows. Suppose there are n existing flows and m new flows; the flow controller first calculates the updated C_{flow} (line 2). It then redistributes credits from the current flows to the new flows until each new flow receives its allocated C_{flow}. Two scenarios arise during this process: (a) If $c_i \geq \frac{m}{n} C_{flow}$, the flow controller can immediately transfer $c_i - \frac{m}{n} C_{flow}$ credits to the new flows (lines 3-6); (b) If $c_i < \frac{m}{n} C_{flow}$, the flow controller records the under-allocated flows in I (line 8), transfers their credits to prevent starvation of newly arrived flows (lines 10-14), and replenishes owed credits when credits are released (lines 19-25).

• **Q2: Handling slow path flows.** We provide two strategies for invoking the handler to process slow path I/O flows with transparency guarantee: a CEIO driver API for proactive invocation by CPUs and event-driven invocation by NIC cores (*e.g.*, triggered upon receiving an RDMA Write-with-immediate message). The former, being more efficient, is used by default for CPU-involved flows, while the latter, offering greater flexibility, is default for CPU-bypass flows. Since the flow controller does not allocate credits for packets in the slow path, DMAing these packets to host memory will flush the LLC. To mitigate this, the flow controller temporarily pauses the fast path during slow path DMAing, drains the I/O flow, and then re-enables the fast path. Additionally, CCA is triggered when NIC cores detect that the network's production rate exceeds the consumption rate of the CPU or memory controller in the slow path. This approach not only ensures that packets in the slow path are drained efficiently but also allows flows degraded to the slow path to upgrade back to the fast path, thereby maintaining fairness.

• **Q3: Reallocation and scaling.** We adopt an "active flow" strategy combined with a round-robin scheduling policy to reallocate credits, making CPU-involved flows more likely to access the fast path, and scaling to large number of flows. Identifying inactive flows is straightforward: (a) for some long time inactive flows, a simple timer can be used to coarsely identify them (*e.g.*, 1 second); (b) for others, we consider that if a flow's packets in the fast path remain unprocessed until the slow path draining is proactively invoked, it is likely to be inactive. The flow controller then recycles the credits from inactive flows and reallocates them using Algorithm 1. Additionally, as a backup strategy, we use a timer to periodically re-activate inactive flows in round-robin, ensuring fairness and the opportunity for all flows to access fast path resources.

4.2 Elastic Buffering

The primary design goal of elastic buffering is to minimize interaction overheads, introduced by packet ordering and synchronous access to on-NIC memory. We now describe how these challenges are addressed respectively.

Eliminating Out-of-Order via Software Ring. Under our flow control design, we observe that, although packets may reside in both fast path and slow path buffers, packets in the fast path always reach the LLC before those in the slow path. Specifically, the fast path is continuously utilized until the flow exhausts its credits. Once the flow is degraded to the slow path, the fast path is blocked until the packets in the slow path are drained. This alternation between the fast and slow paths gives an opportunity to facilitate the order processing in CPUs. We implement a CEIO software ring (SW ring) to abstract the hardware rings (*i.e.*, fast path ring and slow path ring) for upper-layer applications, providing an ordering reception interface (`recv()`) shown in Figure 7. The SW ring unifies fast-path and slow-path packets into a single application-facing abstraction. This ring is implemented as a two-producer, one-consumer buffer where its tail pointer is updated alternately based on the tail pointers of the HW rings (hardware rings). Similar to the legacy ring buffers, the SW head pointer is incremented immediately after transferring packets' ownership to the application, while HW head pointers are incremented after previous packets have been processed by the application (similar to RDMA's posted work requests). To ensure ordering correctness across path transitions, CEIO enforces phase exclusivity. When a flow transitions from fast to slow path, the fast-path DMA is paused

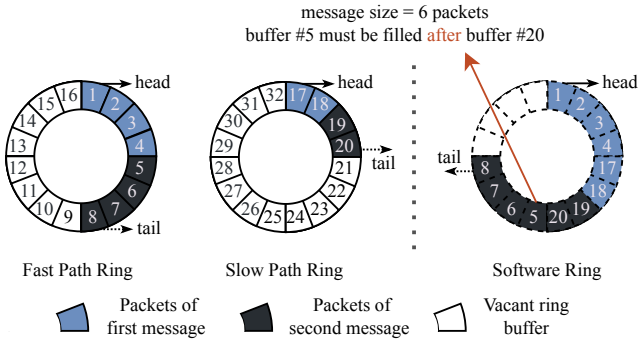


Figure 7: An example of elastic buffering.

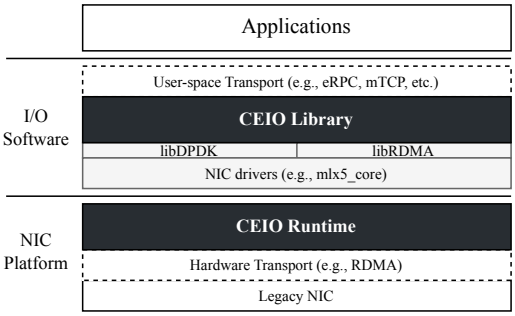


Figure 8: The implementation details and placement of CEIO within the I/O stack.

immediately, and all buffered slow-path packets are drained before the fast path resumes. This guarantees that all fast-path packets are delivered before any slow-path packets for the same flow. Since the packet order is naturally preserved within each hardware ring, all packets in the SW ring are inherently ordered. As a result, CEIO avoids per-packet metadata tracking or sorting commonly seen in software reordering schemes. Additionally, we observe that the lazy credit release mentioned in §4.1 is naturally achieved—fast path's head pointer is updated after processing messages, which triggers NIC cores to replenish the credits of the flow.

Mitigating synchronous access via asynchronous DMA. Note that packets in the slow path are still located in on-NIC memory, meaning the application must wait for DMA operations to complete when accessing these packets. To overlap the synchronous access overhead, we design a non-blocking (`async_recv()`) API (§5) that enables asynchronous fetching of packets from the slow path. Leveraging the alternation between fast and slow paths, CEIO driver can issue DMA read requests for slow path packets while the application processes fast path packets asynchronously. Specifically, the CEIO driver maintains a flag for each ring entry, indicating whether the I/O buffer locates in the fast path or the slow path. While application polls the SW ring, the driver asynchronously issues DMA read requests for slow path packets by checking the flag field. This ensures the application is not blocked because the driver will not wait for DMA completion (similar to DPDK polling mechanism). Then, in the next iteration, the application can directly fetch packets from host memory instead of on-NIC memory[4].

Taking Figure 7 as an example to illustrate elastic buffering. Suppose 2 messages arrive, and 4 credits remain. According to CEIO, the first 4 packets will be filled into the fast path ring (I/O buffers #1-#4), and the subsequent packets will be DMAed to on-NIC memory and filled into the slow path ring. Consider the NIC just receives #20, and the application requires the first message (I/O buffers #1-#4 and #17, #18). The driver first fetches the packets in the fast path and issues DMA read requests for #17 and #18. For the synchronous API (`recv()`), the driver returns to the application after completing the DMA. For the asynchronous API (`async_recv()`), the driver immediately returns with buffers #1-#4. Meanwhile, since buffers #17 and #18 are located in the slow path, the buffer manager continuous to DMA subsequent #19 and #20 to the host memory, draining the slow path. Once finishing, the flow controller re-enables the fast path, allowing the remaining packets to be filled into buffers #5-#8 of the fast path ring. When calling the second message, CEIO driver repeats the above process and maintains the order of packets in the SW ring (fill #19-#20 first, then #5-#8).

[4]A minor detail is that the slow path head pointer is updated immediately after the DMA operation completes, providing two key benefits: (a) reducing pointer update overhead for Memory-mapped I/O (MMIO) interaction; (b) enabling the driver to track which DMA operations have completed.

5 Implementation

We have fully implemented CEIO on commercially available SmartNICs, the NVIDIA BlueField-3 DPU with DOCA SDK v2.9.2 [51]. In this section, we demonstrate CEIO's compatibility to existing I/O frameworks that ensures the transparency of the credit-based flow control and the elastic buffering. Figure 8 illustrates CEIO's placement in the I/O stack, where a CEIO library is exposed to applications and a CEIO runtime is deployed in the NIC.

Softeware Compatibility. We have developed CEIO over DPDK v22.10 and MLNX_OFED v23.10 to support both DPDK and RDMA based I/O frameworks. This is achieved by CEIO library, built on top of `librte_ethdev` and `libibverbs`, and exposing socket-like blocking (`recv()`) and non-blocking (`async_recv()`) APIs to applications. The transport layer operates either above CEIO (for user-space stacks) or beneath it (for hardware-based transports). As a result, CEIO serves as an enhancement for the conventional firmware/driver layer, while maintaining fully compatible with existing transport protocols since CEIO does not modify transport layer behavior. This feature enables CEIO to natively support most existing user-space networking libraries, such as mTCP [35] and Open vSwitch [59]. We also believe that integrating CEIO into kernel-space libraries, such as XDP [27] is feasible and will be explored in future work. Note that CEIO is deployed only on the receiver side, while the sender side continues to running with the original networking libraries.

Next, we use eRPC [37], a popular RPC framework that supports both DPDK and RDMA network interfaces, to demonstrate CEIO's software compatibility more concretely. To utilize CEIO, users only need to replace the low-level I/O receiving operations [30] with CEIO's `recv()` and `async_recv()` APIs. CEIO library will handle all remaining operations described in §4.1 and §4.2, including the management for CEIO fast and slow path rings, credit replenishment, and ownership transfer of I/O buffers to the eRPC. Additionally, we provide zero-copy I/O support by implementing `post_recv()` API, which allows the application to allocate and transfer the ownership

of a memory buffer to CEIO driver, and CEIO will utilize the buffer as an I/O buffer for subsequent DMA operations.

Hardware Compatibility. CEIO leverages the RMT engine and ARMv8 on-NIC cores in the BlueField-3 DPU to implement the flow controller and elastic buffer manager. We believe that this hardware dependency does not pose a significant constraint, as the core functionality of CEIO can be easily implemented on other brand of SmartNICs, whether they use off-path core, on-path core, or ASIC/FPGA-based architectures [44]. Specifically: (a) the RMT engine is a standard feature in most SmartNICs [4, 33] and legacy NICs [21, 52], supporting basic RSS [71] and ARFS [72] offloading; and (b) CEIO logic in ARMv8 core mainly consists of match-action, DMA, and queue management, of which the atomic operations are lightweight table lookup and register access. Even a wimpy processor in on-path NIC architecture [7, 46] is sufficient to implement CEIO's logic, as the memory access overhead is negligible. Additionally, we have observed that the required atomic operations are common in some ASIC/FPGA-based NICs [7, 21].

In CEIO, elastic buffering is implemented using the on-board DRAM of SmartNICs or DPUs, as this memory is software-accessible. If SRAM on legacy NICs such as ConnectX-7 or ConnectX-8 were exposed for general-purpose use, this limited capacity would still be adequate for CEIO's functionality. Accordingly, CEIO does not aim to inflate NIC resources, but rather prefers a minimal and already-available memory footprint to improve I/O efficiency.

6 Evaluation

We evaluate CEIO with real-world workloads under various network conditions. In general, CEIO delivers near line-rate throughput and µs tail latency in most cases, outperforming SOTA methods by up to 2.5× in throughput and 1.9× in P99.9 tail latency. Our goals in this section are:

- Understand benefits of CEIO's core idea—an I/O manager in the NIC with proactive rate control and elastic buffering. By testing end-to-end performance improvements, we evaluate whether CEIO addresses the fundamental limitations of previous LLC optimizations, such as HostCC and ShRing, through experiments conducted under dynamic flow distribution and network burst (§6.2).
- Deep dive into CEIO's micro-behaviors—including performance in the fast path, slow path, thousands of flows, and mixed I/O flows—to show the effectiveness of CEIO's design (§6.3).
- Derive lessons from CEIO's evaluation. These lessons provide guidelines for the optimal usage of CEIO and inform future designs for I/O software and hardware (§6.4).

6.1 Methodology

The experiments are conducted on two 200Gbps servers, as described in §2.3. One server hosts an eRPC-based key-value server [37] and a LineFS DFS server [36], while the other is configured as the client. We use the throughput and P99.9 latency of eRPC and LineFS as baselines. Since eRPC is built on both DPDK and RDMA, and LineFS is built on RDMA, we evaluate CEIO in both DPDK and RDMA configurations. We then evaluate two cache optimizations for I/O discussed in §2.3: the reactive I/O rate control method HostCC [1] and the fixed I/O buffer method ShRing [61], using them as competitive baselines. Each method is configured as follows:

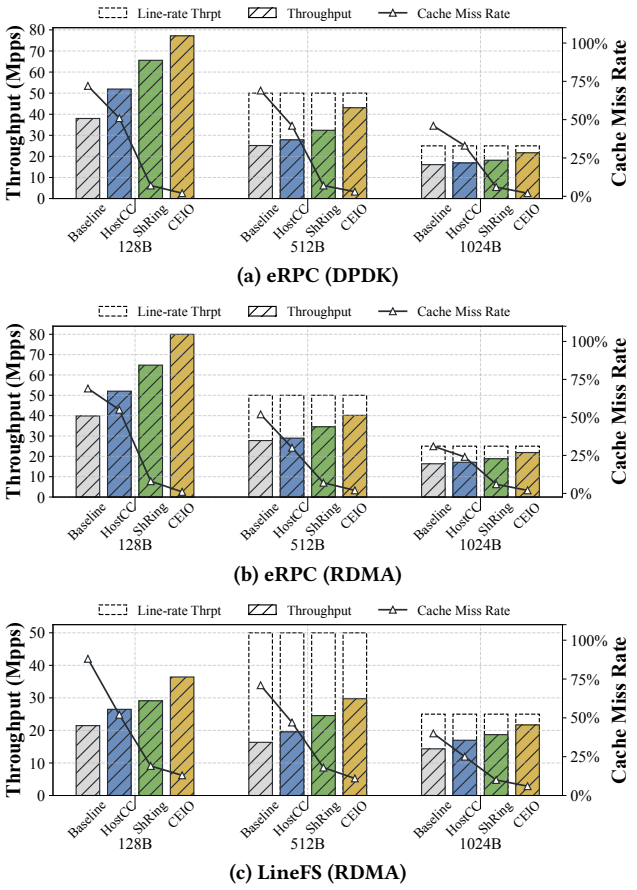


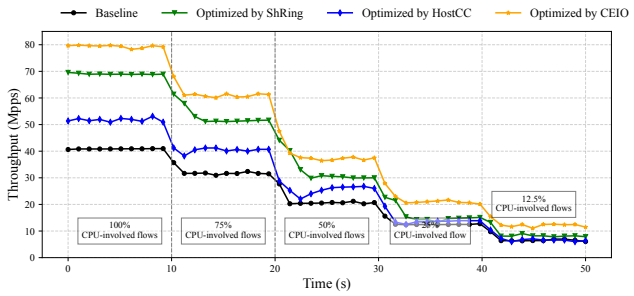
Figure 9: Throughput and LLC miss rate of CEIO and other benchmarks with various packet sizes (128B-1024B). CEIO significantly reduces the LLC miss rate and achieves higher throughput under static network conditions.

- HostCC is deployed as a kernel module, configured to monitor IIO buffer occupancy and PCIe bandwidth utilization. It dynamically adjusts host resource allocation (e.g., CPU processing time, PCIe credits) for each I/O flow and triggers existing network CCAs (e.g., DCTCP) when host congestion is detected.
- The hardware requirement for ShRing (i.e., shared receive queue) is available in our BlueField-3 NICs [55]. To leverage ShRing, we re-implement the dispatching logic of eRPC and LineFS to use the shared receive queue and limit the number of Rx entries to 4096, which is lower than the LLC capacity (12MB).

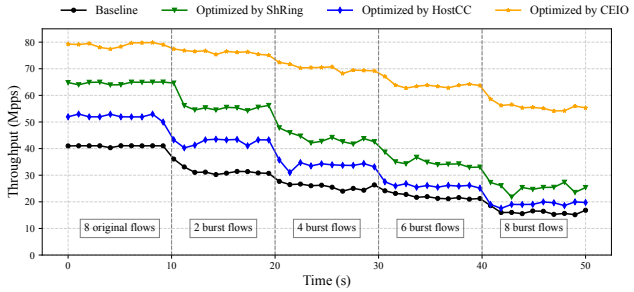
Additionally, we compare the data path performance of CEIO with a popular RDMA benchmark, `perftest`[54] developed by Mellanox, to demonstrate that our optimizations for mitigating slow path overhead are effective and efficient.

Benchmarking Applications. We evaluate CEIO using the following real-world workloads:

• Key-value operations on eRPC: The server handles 1:1 get/put requests with an 1:4 key-value ratio (e.g., 16B key, 64B value, resulting in a 144B packet). We populate 1,000 key-value entries and generate requests randomly from 8 client threads, aiming to saturate the server node and network fabrics.



(a) I/O performance in dynamic flow distribution.



(b) I/O performance in network burst.

Figure 10: I/O performance of CEIO and other benchmarks in various network conditions. CEIO avoids the fundamental limitations of the SOTA optimizations.

• File transfer on LineFS: The client writes a 16GB file to the server in different chunk sizes, while the server performs replication and logging. 8 client threads perform the write operations in parallel to ensure that the client does not become a bottleneck.

• Echo: One client continuously sends messages to the server, which echoes each message back with a 64B acknowledgement. This workload is used to demonstrate the highest performance of CEIO's I/O data path. We use dperf [31] to generate the echo requests.

6.2 End-to-End Performance

We report the end-to-end performance improvements of CEIO compared to the baselines in terms of throughput and latency, and benchmark CEIO against SOTA optimizations. Results are presented in Figure 9, Table 1, and Figure 10, with following observations:

CEIO vs. Baselines. CEIO achieves a throughput speedup of 1.3-2.1× and a latency reduction of 2.0-4.7× over the baseline. These speedups are attributed to optimized LLC utilization, where CEIO reduces the LLC miss rate from 88% to 1%. This observation aligns with the analysis in §3, where CEIO's credit-based flow control effectively minimizes frequent LLC misses. Additionally, the speedups are more pronounced for eRPC, as eRPC employs a zero-copy optimization for packet processing, significantly reducing CPU cycles and allowing it to achieve higher theoretical throughput compared to LineFS. We also observe that CEIO delivers similar performance gains in both DPDK and RDMA configurations, demonstrating its compatibility across different setups.

CEIO vs. HostCC and ShRing. We first compare CEIO with HostCC and ShRing under static network conditions, as shown in Figure 9.

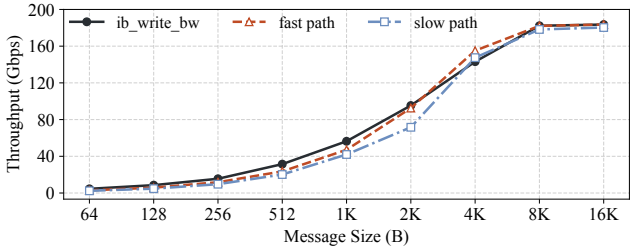


Figure 11: Throughput comparison between CEIO fast path, slow path, and ib_write_bw in perftest. The overhead of CEIO's flow control is negligible.

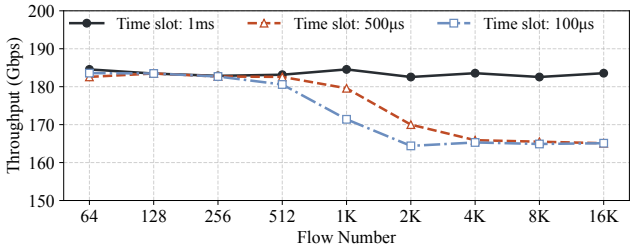


Figure 12: Aggregate throughput of CEIO with 512B-echo workload in RDMA UD mode, varying the number of flows. The results indicate CEIO's active flow strategy helps sustain throughput under high flow count.

We observe that (a) the proactive rate control in CEIO is more effective than the reactive rate control in HostCC, resulting in up to 1.5× throughput speedup and a 55× improvement in cache hit rate; (b) although the cache miss rates of CEIO and ShRing are similar, CEIO achieves higher throughput due to elastic buffering, which prevents unnecessary CCAs triggered by packet loss. We then repeat the experiments under dynamic network conditions, as described in §2.3. When we intensify the limitations of HostCC and ShRing, CEIO performs better, achieving up to 2.0× throughput speedup in dynamic flow distribution (Figure 10a) and up to 2.9× throughput speedup in network burst conditions (Figure 10b).

6.3 CEIO Effectiveness

In this section, we take a deep dive into CEIO to: (1) evaluate the performance of CEIO in both the fast path and slow path, assessing its I/O data path efficiency; (2) analyze the scalability of CEIO with respect to the number of flows; and (3) demonstrate the effectiveness of credit-based flow control and elastic buffering in mitigating slow path overheads.

Performance in Fast Path and Slow Path. First, we report the single-flow performance of CEIO in both the fast path and slow path using the echo workload, where we force the flow into the slow path by setting its credit to zero. To demonstrate CEIO's high efficiency, we also conduct experiments using the ib_write_bw tool in perftest, a standard RDMA benchmark. The results, demonstrated in Figure 11, reveal the following observations: (1) CEIO achieves a similar performance curve in the fast path, indicating that the overhead of credit-based flow control is negligible; (2) the slow path performance approaches the fast path once the message size exceeds 4KB, with the performance gap remaining under 22%.

Datapaths	eRPC(DPDK)		eRPC(RDMA)		LineFS	
	P99	P99.9	P99	P99.9	P99	P99.9
Baseline	116.87	126.94	137.31	184.03	276.29	432.81
HostCC	102.14 (↓ 1.14×)	107.39 (↓ 1.18×)	121.58 (↓ 1.13×)	169.45 (↓ 1.09×)	154.28 (↓ 1.79×)	283.36 (↓ 1.53×)
ShRing	64.05 (↓ 1.82×)	85.42 (↓ 1.49×)	94.82 (↓ 1.45×)	137.36 (↓ 1.34×)	121.48 (↓ 2.27×)	162.50 (↓ 2.66×)
CEIO	46.19 (↓ 2.53×)	53.08 (↓ 2.39×)	69.50 (↓ 1.98×)	73.70 (↓ 2.50×)	66.28 (↓ 4.17×)	91.43 (↓ 4.73×)

Table 2: P99 and P99.9 latency (µs) of CEIO and baseline approaches under the 512B echo workload. The symbol ↓ denotes the latency reduction factor

	RDMA Write	Fast Path	Slow Path
64B	1.46	1.72	1.80
1024B	2.10	2.31	2.43
4096B	2.54	3.32	3.76

Table 3: Latency metrics (µs) of CEIO in the fast and slow paths, tested with ib_write_lat tool in perftest. The baseline is the latency of RDMA write.

Ratio	Baseline	CEIO w/o optimization	CEIO
3:1	31.678 (-)	45.299 (1.53×)	61.570 (1.94×)
1:1	20.521 (-)	28.319 (1.38×)	37.373 (1.82×)
1:3	12.429 (-)	14.418 (1.16×)	21.305 (1.71×)

Table 4: Throughput (Mpps) of CPU-involved flows and speedup of CEIO with and without fast/slow path optimizations on mixed I/O flows. The ratio refers to the ratio of CPU-involved to CPU-bypass flows. There are 8 flows in total.

We then report the latency metrics of CEIO at Table 3. The results show that CEIO does introduce modest latency overhead (1.10× - 1.48×) due to the additional controlling logic (§4). This suggests that users may need to adjust time-sensitive thresholds in their transport protocols, such as retransmission timeouts and RTT estimates in CCAs. However, since the absolute latency overhead is less than 10µs, which is several orders of magnitude smaller than typical transport protocol constants, we believe the impact on transport performance is negligible. Additionally, we observe that the slow path exhibits significantly higher latency than the fast path under large packet conditions, primarily due to data traversing the lower-performance onboard memory. We believe this latency overhead can be further reduced by implementing CEIO within the NIC hardware pipeline (e.g., using NVIDIA DPA and onboard SRAM), which we leave as future work.

Performance in Thousands of Flows. Next, we analyze how the active flow optimization in §4.1 scales CEIO to support thousands of flows. To eliminate the negative impact of native scaling overheads discussed in DPDK [24] and RDMA [77], we utilize RDMA Unreliable Datagram (UD) mode to generate flows. Specifically, the client concurrently sends 16 flows with different queue pair IDs, maintains a short time slot, and randomly changes the destination queue pairs for each subsequent time slot. We report the aggregated throughput of CEIO as the number of flows varies in Figure 12 and observe the following: (1) CEIO maintains stable, high throughput when the flow changing rate is slow (e.g., when the time slot is set above 1ms); (2) When the flow changing rate is fast (e.g., when the time slot is set to 500 and 100 µs), CEIO experiences a slight throughput decrease from 128 to 1K flows, and drops to the slow path performance beyond 1K flows. This is because the simple round-robin re-activation strategy cannot keep up with the flow changing rate, causing all flows to shift to the slow path. We leave this challenge for future work.

Performance in Mixed I/O Flows. Finally, we evaluate the performance of CEIO in mixed I/O flows to demonstrate the effectiveness of our designs. To this end, we deploy eRPC and LineFS together and generate CPU-involved and CPU-bypass flows with varying ratios. The flow characteristics remain consistent with those in Figure 10. The results, presented in Table 4, reveal the following observations: (1) When CPU-involved flows dominate, CEIO with optimizations can further improve throughput from 1.53 to 1.94×. This improvement stems from credit reallocation (§4.1), which effectively detects CPU-bypass flows, reallocates credits to CPU-involved flows, and enhances fast path utilization; (2) When CPU-bypass flows dominate, CEIO with optimizations improves throughput from 1.16 to 1.71×. This gain results from the SW&HW ring design and asynchronous access mechanism (§4.2), which effectively mitigate out-of-order and synchronization overheads in the slow path.

Scenarios where CEIO's Benifits are Limited. While CEIO demonstrates improvements across a range of workloads with limited overhead, we also identify scenarios where LLC efficiency is not the dominant factor limiting I/O performance. In such cases, CEIO and other methods provide limited benefit:

• Scenarios with low memory pressure: When the memory footprint is small, almost all I/O data can be cached in the LLC, resulting in negligible cache miss rates. In this case, baseline, CEIO, and other optimizations exhibit similar performance. To demonstrate this, we evaluate a synthetic workload consisting of 64B packets with VxLAN decapsulation, and observe that both baselines and CEIO achieve 89Mpps throughput with <5% cache miss rate.

• Scenarios with large packet size: Figure 9 suggests that when packet sizes are large, CEIO offers limited throughput gains. This is because large packets amortize the per-packet overhead (such as ring and memory pool management, descriptor transfer, header overhead, etc.), reducing CPU utilization and thus allowing the system to tolerate the additional memory access overhead from cache misses. To validate this, we evaluate the baseline with 9000B jumbo frames in an echo workload, and observe that it can achieve line-rate throughput when packet size is larger than 4096B, even with a 48% cache miss rate.

6.4 Lessons Learned

The above evaluation demonstrates that CEIO has the ability to achieve near line-rate throughput and µs latency for the I/O data path. However, it is not a universal solution for all scenarios. We highlight two cases where CEIO's performance is suboptimal and provide insights for the future I/O software and hardware designs.

Understanding Improvement Gaps between eRPC and LineFS. In Figure 9, we observe that eRPC with CEIO delivers near line-rate throughput, while LineFS with CEIO achieves only 45% of eRPC's throughput at the worst point. This suggests that zero-copy optimizations, like those employed by eRPC, are essential for achieving high throughput for two main reasons: (1) zero-copy saves more CPU cycles for packet processing, as discussed in §6.2; (2) memory copy operations in LineFS typically result in additional cache misses, with a observed miss rate of around 10%, even with an optimized I/O data path. Specifically, when copying I/O data from the I/O buffer to the application buffer, the latter is usually not in the LLC cache, leading to further cache misses. This observation underscores the necessity of zero-copy for unlocking the full potential of CEIO, and suggests that future network libraries should consider cache optimization techniques, such as recycling and pre-fetching application buffers from a memory pool, to further enhance performance.

Understanding Performance Penalties of Slow Path. By comparing Figure 11 with Figure 12, we observe that the slow path performance drops by around 15Gbps when the number of flows is large and the message size is small (e.g., 512B). We suspect this degradation is due to two factors: (1) increased latency in processing DMA requests from on-NIC memory to DRAM, caused by the internal PCIe switch of the BlueField-3 [78]; and (2) degraded on-NIC memory throughput due to chaotic access patterns for the on-NIC memory [65]. To address these issues, we suggest that future NIC architectures allocate CPU-attached SRAM (such as those in the CXL architecture [66]), bypassing the internal PCIe switch, to further reduce synchronization overhead in CEIO's slow path.

Understanding the Relationship between Transport Protocols and CEIO. CEIO operates independently of specific transport-layer protocols such as DCTCP [2] and RoCE [26]. While transport protocols focus on end-to-end congestion control and loss recovery—typically reacting to network conditions over tens or hundreds of microseconds—CEIO provides fast, local response to host-side memory congestion, which can be considered as a "flow control" mechanism, from NICs to CPUs. We believe these two approaches are complementary and should coexist to ensure both host and network-level efficiency.

7 Related Work

This section reviews related LLC optimization works beyond the I/O system, as well as related I/O acceleration techniques that are orthogonal to CEIO.

Software-level Optimization. PacketMill [18] reserves a portion of the cache specifically for DDIO. Shenango [58] adopts a prefetching strategy to warm up the LLC before processing packets. ScaleRPC [13] limits concurrent flows to mitigate cache contention. CEIO can be integrated with these works to further improve the LLC efficiency as CEIO operates transparently to upper-layer software.

Hardware-level Optimization. Since LLC plays a crucial role in I/O performance, some hardware-level works focus on expanding LLC capacity [19] or re-designing the cache allocation strategy for DDIO [84]. We believe that CEIO's design remains effective in conjunction with these efforts, as the basic assumption, *i.e.*, the cache is shared by multiple I/O flows and can easily be exhausted by I/O traffic, is still valid.

Novel I/O Accelerators. We have observed that some works propose new I/O acceleration techniques beyond DDIO and RDMA, such as optimizing address translation [77] and ring buffer management [65]. These techniques can be integrated into CEIO by replacing other components within the I/O data path, such as the RDMA engine and SRAM controller, thereby constructing a more complete, next-generation NIC architecture.

8 Conclusion

This paper presents CEIO, a cache-efficient I/O architecture for NIC-CPU data paths. CEIO introduces proactive, credit-based I/O rate control and elastic buffering to eliminate LLC misses in the I/O data path, optimizing the effectiveness of DDIO and RDMA under varying network conditions. We implement CEIO using commodity SmartNICs and expose libraries compatible with DPDK and RDMA. Extensive evaluations demonstrate CEIO's ability to achieve near line-rate throughput and µs-scale latency across various scenarios, outperforming state-of-the-art solutions such as HostCC and ShRing by up to 2.9× in throughput and 1.9× in latency.

Acknowledgments

We thank the anonymous SIGCOMM reviewers and our shepherd, Gianni Antichi, for their insightful comments. This work is supported in part by the Hong Kong RGC TRS T41-603/20R, ITC ACCESS, TACC [82], HKUST-Inspur Cloud joint research project, the China NSFC 62402407, and the State Key Laboratory Open Project KFKT2025B27. Kai Chen is the corresponding author.

References

[1] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. 2023. Host Congestion Control. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 275–287.

[2] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*. 63–74.

[3] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 435–446.

[4] AMD. 2024. AMD Pensando™ Pollara 400 Adapters. https://www.amd.com/en/products/accelerators/pensando.html. Accessed 2025-01-01.

[5] Thomas E Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N Schuh, and Emmett Witchel. 2020. Assise: Performance and availability via client-local {NVM} in a distributed file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1011–1027.

[6] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Weicheng Sun. 2014. PIAS: Practical information-agnostic flow scheduling for data center networks. In *Proceedings of the 13th ACM workshop on hot topics in networks*. 1–7.

[7] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2022. hXDP: Efficient software packet processing on FPGA NICs. *Commun. ACM* 65, 8 (2022), 92–100.

[8] ARM. ARM cache stashing. 2017. https://developer.arm.com/documentation/102407/0100/Cache-stashing. Accessed 2025-01-01.

[9] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 65–77.

[10] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. 2022. Towards μ s tail latency and terabit ethernet: disaggregating the host network stack. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 767–779.

[11] Devon Callahan, Timothy Curry, Hazel Davidson, Heytem Zitoun, Benjamin Fuller, and Laurent Michel. 2023. FASHION: Functional and Attack Graph Secured HybrId Optimization of Virtualized Networks. *IEEE Transactions on Dependable and Secure Computing* 20, 04 (2023), 3093–3109.

[12] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, and Zeke Wang. 2024. Demystifying datapath accelerator enhanced off-path smartnic. *arXiv preprint arXiv:2402.03041* (2024).

[13] Youmin Chen, Youyou Lu, and Jiwu Shu. 2019. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–14.

[14] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. 2017. drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 1–14.

[15] Intel Corporation. Intel data direct i/o technology (intel DDIO): A primer. 2012. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf. Accessed 2025-01-01.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 401–414.

[17] Haitao Du, Yuhan Qin, Song Chen, and Yi Kang. 2024. FASA-DRAM: Reducing DRAM Latency with Destructive Activation and Delayed Restoration. *ACM Transactions on Architecture and Code Optimization* 21, 2 (2024), 1–27.

[18] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. 2021. PacketMill: toward per-Core 100-Gbps networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1–17.

[19] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. 2019. Make the most out of last level cache in intel processors. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–17.

[20] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. 2020. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 673–689.

[21] Alex Forencich, Alex C Snoeren, George Porter, and George Papen. 2020. Corundum: An open-source 100-gbps nic. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 38–46.

[22] Linux Foundation. 2024. DPDK: Data plane development kit. http://dpdk.org. Accessed 2025-01-01.

[23] Linux Foundation. 2025. RDMA-Core github repository. https://github.com/linux-rdma/rdma-core. Accessed 2025-01-01.

[24] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Íñigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. 2024. Making kernel bypass practical for the cloud with Junction. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 55–73.

[25] Google. 2025. gRPC github repository. https://github.com/grpc/grpc. Accessed 2025-01-01.

[26] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 202–215.

[27] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies*. 54–66.

[28] Wentao Hou, Jie Zhang, Zeke Wang, and Ming Liu. 2024. Understanding Routable PCIe Performance for Composable Infrastructures. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 297–312.

[29] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. 2005. Direct cache access for high bandwidth network I/O. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 50–59.

[30] The implementation of low-level I/O operations in eRPC. Line 110-141. https://github.com/erpc-io/eRPC/blob/master/src/transport_impl/dpdk/dpdk_transport_datapath.cc. Accessed 2025-01-01.

[31] Baidu Inc. 2025. dPerf github repository. https://github.com/baidu/dperf. Accessed 2025-01-01.

[32] Intel. 2021. Intel Xeon Silver 4309Y Specification. https://www.intel.com/content/www/us/en/products/sku/215275/intel-xeon-silver-4309y-processor-12m-cache-2-80-ghz/specifications.html. Accessed 2025-01-01.

[33] Intel. 2024. ntel® Infrastructure Processing Unit (Intel® IPU). https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html. Accessed 2025-01-01.

[34] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 947–960.

[35] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 489–502.

[36] Kim Jongyul, Jang Insu, Reda Waleed, Im Jaeseong, Canini Marco, Kostic Dejan, Kwon Youngjin, Peter Simon, and Witchel Emmett. 2021. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. 756–771.

[37] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPC can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 1–16.

[38] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2014. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*. 295–306.

[39] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 437–450.

[40] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. FaSST: fast, Scalable and Simple Distributed Transactions with Two-Sided RDMA Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 185–201.

[41] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Michael Kalia, and Anuj Kaminsky, David G. Andersen, Seongil O, Sukhan Lee, and Pradeep Dubey. 2016. Full-stack architecting to achieve a billion-requests-per-second throughput on a single key-value store server platform. *ACM Transactions on Computer Systems (TOCS)* 34, 2 (2016), 1–30.

[42] Wenxin Li, Xin He, Yuan Liu, Keqiu Li, Kai Chen, Zhao Ge, Zewei Guan, Heng Qi, Song Zhang, and Guyue Liu. 2024. Flow scheduling with imprecise knowledge. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 95–111.

[43] Wenxue Li, Xiangzhou Liu, Yuxuan Li, Yilun Jin, Han Tian, Zhizhen Zhong, Guyue Liu, Ying Zhang, and Kai Chen. 2024. Understanding communication characteristics of distributed training. In *Proceedings of the 8th Asia-Pacific Workshop on Networking*. 1–8.

[44] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*. 318–333.

[45] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. 2024. Serialization/Deserialization-free State Transfer in Serverless Workflows. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 132–147.

[46] Marvell. 2020. Marvell LiquidIO™ III. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf. Accessed 2025-01-01.

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

[48] NVIDIA. 2020. https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3090-3090ti/. Accessed 2025-01-01.

[49] NVIDIA. 2022. https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/rtx-4090/. Accessed 2025-01-01.

[50] NVIDIA. 2024. https://docs.nvidia.com/cuda/gpudirect-rdma/. Accessed 2025-01-01.

[51] NVIDIA. 2024. DOCA Software Development Kit v2.9.2 LTS. https://docs.nvidia.com/doca/archive/2-9-2-lts-ovs-update/index.html. Accessed 2025-01-01.

[52] NVIDIA. 2024. NVIDIA ConnectX-7 Network Adapter. https://docs.nvidia.com/networking/display/mlnxofedv24100700/flow+steering. Accessed 2025-01-01.

[53] NVIDIA. 2024. OVS Offload Using ASAP Direct. https://docs.nvidia.com/networking/display/mlnxofedv24100700/ovs+offload+using+asap2+direct. Accessed 2025-01-01.

[54] NVIDIA. 2024. Performance Tuning for Mellanox Adapters. https://enterprise-support.nvidia.com/s/article/performance-tuning-for-mellanox-adapters. Accessed 2025-01-01.

[55] NVIDIA. 2024. Shared Receive Queues in BlueField-3 Platform. https://docs.nvidia.com/networking/display/bluefielddpuosv452/shared+rq+mode. Accessed 2025-01-01.

[56] NVIDIA. 2025. NCCL github repository. https://github.com/NVIDIA/nccl. Accessed 2025-01-01.

[57] OpenMPI. 2025. OpenMPI github repository. https://github.com/open-mpi/ompi. Accessed 2025-01-01.

[58] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 361–378.

[59] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. 2015. The design and implementation of open vSwitch. In *12th USENIX symposium on networked systems design and implementation (NSDI 15)*. 117–130.

[60] Solal Pirelli and George Candea. 2020. A Simpler and Faster NIC Driver Model for Network Functions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 225–241.

[61] Boris Pismenny, Adam Morrison, and Dan Tsafrir. 2023. ShRing: Networking with Shared Receive Rings. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 949–968.

[62] NVIDIA BlueField Networking Platform. 2025. https://www.nvidia.com/en-us/networking/products/data-processing-unit/. Accessed 2025-01-01.

[63] Redis. 2025. Redis github repository. https://github.com/redis/redis. Accessed 2025-01-01.

[64] Zhenghang Ren, Yuxuan Li, Zilong Wang, Xinyang Huang, Wenxue Li, Kaiqiang Xu, Xudong Liao, Yijun Sun, Bowen Liu, Han Tian, Junxue Zhang, Mingfei Wang, Zhizhen Zhong, Guyue Liu, Ying Zhang, and Kai. Chen. 2025. Enabling Efficient GPU Communication over Multiple NICs with FuseLink. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 91–108.

[65] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S Berger, James C Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. 2023. Ensō: A Streaming Interface for NIC-Application Communication. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 1005–1025.

[66] Henry N Schuh, Arvind Krishnamurthy, David Culler, Henry M Levy, Luigi Rizzo, Samira Khan, and Brent E Stephens. 2024. CC-NIC: a Cache-Coherent Interface to the NIC. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 52–68.

[67] Apache Spark. 2025. Spark github repository. https://github.com/apache/spark. Accessed 2025-01-01.

[68] Dan Tang, Yungang Bao, Weiwu Hu, and Mingyu Chen. 2010. DMA cache: Using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 1–12.

[69] Shelby Thomas, Rob McGuinness, Geoffrey M Voelker, and George Porter. 2018. Dark packets and the end of network scaling. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*. 1–14.

[70] Shelby Thomas, Geoffrey M Voelker, and George Porter. 2018. CacheCloud: Towards Speed-of-light Datacenter Communication. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*.

[71] Microsoft. Introduction to receive side scaling. 2017. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling. Accessed 2025-01-01.

[72] Herbert Tom and de Bruijn Willem. Scaling in the linux networking stack. 2011. https://www.kernel.org/doc/Documentation/networking/scaling.txt. Accessed 2025-01-01.

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

[74] Zirui Wan, Jiao Zhang, Yuxiang Wang, Kefei Liu, Haoyu Pan, Yongchen Pan, and Tao Huang. 2025. RHCC: Revisiting Intra-Host Congestion Control in RDMA Networks. *IEEE Transactions on Networking* 33, 3 (2025), 1189–1202.

[75] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. 2024. Towards Domain-Specific Network Transport for Distributed DNN Training. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1421–1443.

[76] Minhu Wang, Mingwei Xu, and Jianping Wu. 2022. Understanding I/O direct cache access performance for end host networking. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 1 (2022), 1–37.

[77] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, Tianhao Wang, Weicheng Ling, Kejia Huo, Pingbo An, Kui Ji, Shideng Zhang, Bin Xu, Ruiqing Feng, Tao Ding, Kai Chen, and Chuanxiong Guo. 2023. SRNIC: A scalable architecture for RDMA NICs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1–14.

[78] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. Characterizing Off-path SmartNIC for Accelerating Distributed Systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 987–1004.

[79] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 87–104.

[80] Li Wenxue, Zhang Junyi, Liu Yufei, Zeng Gaoxiong, Wang Zilong, Zeng Chaoliang, Zhou Pengpeng, Qiaoling Wang, and Kai Chen. 2024. Cepheus: Accelerating Datacenter Applications with High-Performance RoCE-Capable Multicast. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2024, Edinburgh, United Kingdom, March 2-6, 2024*. IEEE, 908–921.

[81] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Songyuan Sui, Khalid Manaa, Omer Shabtai, Yonatan Piasetzky, Matty Kadosh, Arvind Krishnamurthy, TS Eugene Ng, and Ang Chen. 2023. Unleashing SmartNIC packet processing performance in P4. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 1028–1042.

[82] Kaiqiang Xu, Decang Sun, Hao Wang, Zhenghang Ren, Xinchen Wan, Xudong Liao, Zilong Wang, Junxue Zhang, and Kai Chen. 2025. Design and Operation of Shared Machine Learning Clusters on Campus. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 295–310.

[83] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: A distributed file system for Non-Volatile main memory and RDMA-Capable networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 221–234.

[84] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. 2021. Don't forget the I/O when allocating your LLC. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 112–125.