Albatross: A Containerized Cloud Gateway Platform with FPGA-accelerated Packet-level Load Balancing

Jianyuan Lu[1†], Shunmin Zhu[2,1†△], Jun Liang[1†], Yuxiang Lin[1], Tian Pan[1△], Yisong Qiao[1], Yang Song[1], Wenqiang Su[1], Yixin Xie[1], Yanqiang Li[1], Enge Song[1], Shize Zhang[1], Xiaoqing Sun[1], Rong Wen[1], Xionglie Wei[1], Biao Lyu[1], Xing Li[3,1]
[1]Alibaba Cloud [2]Hangzhou Feitian Cloud [3]Zhejiang University
alibaba_cloud_network@alibaba-inc.com

Abstract

Alibaba Cloud's centralized gateways relied heavily on high-capacity switching ASICs, but the abrupt halt of Tofino chip evolution in Jan 2023 forced us to seek alternatives that can meet the requirements of performance, supply-chain security, code reuse, and resource efficiency. After evaluating multiple options, we developed *Albatross*, our 3rd gen cloud gateway based on FPGA and x86 CPUs. Albatross delivers FPGA-based packet-level load balancing to the host CPUs to prevent CPU core overload, manages large reorder buffers under high-latency jitters (100μs) during complex cloud service processing, and resolves head-of-line (HOL) blocking from packet losses or software exceptions in CPUs. To avoid being overloaded by heavy hitters due to anomalies or attacks, it also implements a two-stage rate limiter for millions of tenants with only 2MB of FPGA memory. To maximize resource utilization, Albatross uses containerization to host multiple gateway instances and designs a BGP proxy to lessen the BGP peering overhead on uplink switches caused by high-density container deployments. After hundreds of man-months of development, a single Albatross node can process 80~120Mpps of cloud network traffic with an average latency of 20μs, reducing gateway and sandbox infra costs by 50%.

CCS Concepts

• **Networks** → **Cloud computing**; **Programmable networks**; **Data center networks**.

Keywords

Packet-level Load Balancing, FPGA, CPU, Containerization

ACM Reference Format:
Jianyuan Lu, Shunmin Zhu, Jun Liang, Yuxiang Lin, Tian Pan, Yisong Qiao, Yang Song, Wenqiang Su, Yixin Xie, Yanqiang Li, Enge Song, Shize Zhang, Xiaoqing Sun, Rong Wen, Xionglie Wei, Biao Lyu, Xing Li. 2025. Albatross: A Containerized Cloud Gateway Platform with FPGA-accelerated Packet-level Load Balancing. In *ACM SIGCOMM 2025 Conference (SIGCOMM '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3718958.3750473

[†]Co-first authors. [△]Co-corresponding authors.

1 Introduction

Alibaba Cloud leverages a gateway-centric network virtualization architecture [35]. The gateway clusters need to manage the immense traffic surges of tens of Tbps, including inter-VPC, VPC-to-IDC, VPC-to-Internet traffic, to serve millions of tenants. Initially, we built the gateway clusters using x86 servers. Although the overall performance could be scaled horizontally, the slow progress of single-core CPU performance, coupled with RSS-based traffic distribution [20], failed to address the performance stability issues [31]. Specifically, heavy-hitter flows from dominant tenants could overload a single CPU core, impacting other tenants' traffic hosted on the same core [27, 34]. The Tofino chip [3, 4], with its per-pipeline processing capability of up to multi-Tbps, brought significant improvements to cloud gateway stability. Additionally, the cost per Tbps of Tofino switches [29] was budget-friendly to cloud vendors. Although the limited on-chip resources of Tofino (*e.g.*, PHV, TCAM, SRAM, and pipeline stages) posed challenges for deploying the complex cloud network services for millions of tenants [26, 31], we addressed them in our Sailfish [31] gateway (Tofino-based) through a series of techniques, such as pipeline folding. We adapted well with Tofino, evolving our network virtualization architecture based on the Tofino 2 and 3 roadmap. Unfortunately, in Jan 2023, Intel announced the discontinuation of Tofino chip development [28]. Given that our Sailfish gateway had fully committed to the Tofino technology stack, we began contemplating how to address the performance vacuum left by Tofino.

When selecting the next-gen gateway solution, our first priority was achieving the high performance needed for cloud-scale gateways after Tofino's unavailability. Secondly, after experiencing Tofino's supply-chain disruption, we become more cautious in selecting network ASICs, hoping to choose ones with high availability and no potential supply-chain risks. Thirdly, over the past two generations of gateways, we have accumulated a substantial amount of code that has been validated in production. We aim to maximize code reuse, avoiding reinventing wheels and accelerating time to deployment. Lastly, in previous gateway deployment, each service team (*e.g.*, VPC and SLB teams) managed their own gateways. Cluster utilization was often low and costs were high. For the new gateway, we aim to achieve more efficient resource utilization and to eliminate fragmented operations by separate teams.

With the above requirements, we explored numerous chips and technical solutions. Initially, we sought ASIC alternatives to Tofino but found no product by early 2023 that could fully replace Tofino in terms of P4 compatibility, compiler stability and chip resources. We also considered an ASIC + DPU solution, which was also used in our edge gateway LuoShen [30], but it increased operational complexity and risked DPU overload due to varying ASIC-to-DPU bandwidth convergence. With the DPU market growing rapidly, we also explored server + DPU solutions to offload the gateway data plane to the DPU. However, DPU is typically designed for server-side applications (*e.g.*, vSwitch [32]) and has ample flow table resources for exact matching but often lacks sufficient LPM resources for gateway route lookups (*e.g.*, VXLAN routing table [31]). Moreover, the highly segmented DPU market also raises our concerns about supply-chain issues similar to Tofino. After considering all these factors, we ultimately chose a solution *Albatross* that uses FPGA to accelerate certain capabilities for the x86 gateway while maximizing the reuse of previous x86 gateway code. FPGA also allows to add more customized features to meet diverse tenant demands.

To address CPU core overload with RSS, we built an NIC pipeline on FPGA to implement packet-level load balancing (PLB), spraying packets across CPU cores at the ingress and reordering them at the egress. Although PLB has been implemented in early network processors [1, 21], Intel's DLB [36], and some DPUs (*e.g.*, Octeon 10 [14]), we are the first in the industry to extend PLB capability to external processors across chips. This makes Albatross highly scalable, as its capacity can grow linearly with the latest Intel or AMD CPUs (*e.g.*, AMD EPYC 9965 [6]). However, it also brings new challenges, including large reordering buffer management triggered by significant latency jitters in CPU processing and reordering buffer Head-of-Line blocking caused by packet loss at the CPU side. Furthermore, we unexpectedly found that due to the huge size of cloud network forwarding tables, the performance gap between PLB and RSS is lower than 1% since Albatross under these two modes experience similar cache hit rate, prompting us to focus more on DRAM optimization.

Although PLB reduces the probability of single-core overload, in extreme cases, an anomaly or attack flow may overload the entire gateway after being load-balanced by PLB. We protect the CPU by identifying abnormal heavy-hitters inside the FPGA. However, allocating a meter table entry for each tenant to enforce rate limiting would exhaust the FPGA's on-chip SRAM. To address this, we designed a two-stage rate limiter that can perform rate limiting for millions of tenants with only 2MB SRAM. Additionally, the sampling and pre-check mechanism are employed to resolve false rate limiting on small flows caused by hash collisions.

To improve gateway utilization, we deploy different gateways as containers on x86 servers, virtualizing and partitioning FPGA NIC resources among containers. Containerization also introduces challenges, such as BGP peering overhead on the uplink switch's control plane due to increased container density. We optimized this with a BGP proxy, which reduces the number of BGP peers greatly.

Our main contributions are listed as follows:

- As a cloud vendor that relies on centralized gateways, we shared our first-hand thoughts and actions on the gateway evolution in the post-Tofino era. According to our information, many cloud providers are following Sailfish to build their cloud networks. The industry also faces challenges due to the performance vacuum left by Tofino.
- We proposed a series of techniques in the data plane, including packet-level load balancing and gateway overload protection, to prevent the risk of single-core overload. Notably, we are the first in the industry to design an approach that opens PLB capability to external chips, enabling the gateway's capacity to scale with the latest server CPUs.
- We are the first in the industry to deploy a cloud gateway through containerization. By using a BGP proxy, we avoided overloading the control plane CPU of the uplink switch due to high-density container deployment.
- Albatross has been deployed in Alibaba for over one year, providing 80~120Mpps throughput and 20μs latency on a 2023 CPU. Through containerization, we reduce the infra costs by 50%. We also provide rich experiences and lessons.

2 Background and Motivation

2.1 Evolution of Alibaba's Cloud Gateways

Alibaba Cloud uses a centralized gateway architecture to host millions of tenants [31, 35]. Initially, to rapidly launch various services, x86 clusters were used for quick gateway deployment. But as services and traffic grew, Tofino chips [13] were introduced to accelerate stateless services. Currently, both types of gateways coexist, but each has faced challenges during their evolution.

1st gen x86-based cloud gateways are versatile but limited by single-core performance and NIC port speed. x86-based gateways offer good programmability to support rapid iterations of cloud network services and can boost gateway cluster's performance via horizontal scaling. However, their limited single-core performance constrains tenant service stability when heavy-hitter flows or incast occur. Based on our experience, the current cloud network services process around 1Mpps per core (using kernel bypass [19]). Even with upgraded hardware with significantly more CPU cores, single-core performance still grows slowly. A single CPU core can easily be overloaded by traffic from certain tenants, impacting other tenants on the same core, *e.g.*, a VM with a gigabit vNIC can generate traffic up to 1.6Mpps under stress test with 64B Ethernet packets, exceeding the single-core performance limit. On multi-core CPUs, RSS [20] hashes traffic to the CPU cores at the flow level. Although processing the same flow on one CPU core avoids out-of-order packets, the CPU will experience load imbalance as most traffic is concentrated in a few large flows.

Besides CPU core overloading, we also found that NIC port overloading can impact tenant service stability, especially on gateways with lower NIC speeds developed earlier. While CPU core overloading affects only some tenant services, NIC port overloading not only impacts the stability of certain services but also globally affects control plane protocol maintenance. For example, some gateways use BGP to interact with upstream switches. However, when the NIC port is congested, it indiscriminately drops packets, including control plane protocol messages, leading to control plane maintenance failures and affecting all services on the gateway.

2nd gen Tofino-based cloud gateways have high throughput but insufficient on-chip resources and programmability. To address the stability issues of x86-based gateways, we developed the hardware gateway Sailfish [31] based on Intel Tofino. The gateway has been deployed in Alibaba Cloud since 2019 and now covers all regions. However, during the operation of Sailfish, we also encountered the following challenges. First and foremost is the on-chip resource issue. Tab. 1 shows the current resource consumption of Sailfish on Tofino. Sailfish uses 4 pipelines, and due to pipeline folding, resource consumption varies between pipelines 0,2 and 1,3. As shown in Tab. 1, pipelines 0,2 are most constrained by PHV, while pipelines 1,3 are limited by SRAM. PHV is used for parsing and storing packet headers, and due to Alibaba Cloud's need to support dozens of network protocols and parse their layers, the demand for PHV is high. Pipelines 0,2, as the gateway entry points, require extensive PHV for protocol parsing. SRAM, used for storing forwarding tables, is also heavily consumed, particularly in pipelines 1,3, which store the VM-NC mapping table for millions of tenants.

Table 1: Tofino's resource consumption by Sailfish

Pipeline0,2			Pipeline1,3		
SRAM	TCAM	PHV	SRAM	TCAM	PHV
69.2%	40.3%	**97.0%**	**96.4%**	66.7%	82.3%

Based on the data above, we conclude that Sailfish's resource consumption is nearing its maximum, which severely limits the evolution of cloud services running on it. Specifically, the resource challenges for adding new services are: 1) New packet headers: With PHV utilization at 97%, adding new headers, such as NSH [33] and Geneve [22], is nearly impossible and results in compilation errors; 2) Large table capacity demand: As SRAM utilization is also high, adding new or large tables becomes very difficult; 3) Long-chained functions: Another issue is adding tables that require long-chained matching. Even if resources are sufficient, if the number of required stages exceeds the total stages on the pipeline, compilation will fail. The deep interdependencies between tables due to complex services make this issue common [30].

Besides on-chip resource limitations, Tofino faces programmability issues compared to CPUs when processing complex services. A typical case is Tofino's lack of self-updating tables. For stateful services like SNAT, the pipeline needs to update itself based on the lookup results after packet processing. However, Tofino's table entries can only be written by the control plane via the runtime API, which prevents self-updating operations. Furthermore, since Tofino does not support timers, table aging has to be done via the control plane. Due to the limitations of on-chip PHV and stage resources, it also struggles to support the parsing of complex nested protocols, such as IP options or Zoonet probing packets [37].

We adapted well with Tofino, but unfortunately, it stopped evolving. Given Tofino's significant performance advantages, we have gradually migrated core services to Sailfish, with thousands of devices now handling Tbps-level traffic. We are also closely monitoring Intel's Tofino roadmap, which includes upcoming Tofino 2 and 3 chips that address the on-chip resource limitations. Additionally, we've built a software-hardware collaborative deployment architecture that uses x86 fallback to overcome Tofino's resource and programmability limitations [31]. As our reliance on Tofino deepens, we have encountered bugs during service deployment and continue to discuss with Intel for iterative improvement. Before 2023, everything was progressing positively.

Unfortunately, good times are short. In January 2023, Intel announced the end of Tofino's evolution [28]. While the exact reasons remain unclear, it likely stems from Tofino chips failing to meet expectations of transforming all switches in data centers through programmable capabilities. While we acknowledge that Tofino chips are awesome, they were only used at overlay tunnel endpoint gateways, and during that process, we replaced lots of x86-based gateways. From the perspective of cloud vendors, we have to consider how to evolve our gateways in the post-Tofino era and how to avoid the risks of being locked in by the sudden halt in the evolution of specialized switching ASICs.

Other issues in previous two generations of cloud gateways. Physical gateway clusters have less than 10% utilization during the initial setup period or serving a small number of customers due to their location. The low utilization will be exacerbated particularly for low-traffic services where backup gateways (at a 1:4 or 1:8 ratio) are used. Similarly, service-specific sandbox clusters, necessary for resilience but lacking elastic scaling, add to the initial cost. Although consolidation of various gateways (like in edge cloud environment) improved overall utilization [30], it introduced other issues like increased blast radius and multi-tenant service interference.

2.2 Options After Tofino Stopped Evolving

Below are the various Plan B options we considered after Intel announced the halt of Tofino's evolution.

High-performance switching ASICs. We considered alternative ASIC chips to Tofino. After researching similar switching ASICs like Broadcom TD4 [10] and Cisco Silicon One Q200 [11], we found that, as of Spring 2023, although these chips offer strong forwarding performance, none could replace Tofino in terms of language friendliness (which is important to reuse our P4 code), compiler environment stability, and chip resource specifications. For example, Broadcom TD4 requires NPL programming, has a limited number of stages, and does not support pipeline folding. Cisco Silicon One Q200 has much fewer meter table resources, hindering effective multi-tenant bandwidth throttling, and its P4 programming model differs from Tofino's native P4.

Switching ASICs + DPUs. We also explored the solution of combining DPUs having richer functions with switching ASICs, using a front-end ASIC to distribute traffic to the DPUs, and leveraging multiple heterogeneous chips to meet different processing needs for diverse service scenarios. This solution balances forwarding performance with rich service support and can evolve based on our hyper-converged edge cloud gateway architecture [30]. However, this heterogeneous architecture introduces higher control and forwarding plane complexity, making maintenance more difficult. Additionally, the bandwidth convergence ratio between ASIC and DPUs varies across different service scenarios, leading to issues where ASIC directs large traffic to overload the DPUs.

Server-based partial offloading to DPUs. We noticed a surge in the market of feature-rich DPUs in recent years, such as BF3 [15], Intel IPU [12], AMD Pensando [7], etc. Although they are invented initially for server-side acceleration (*e.g.*, OVS offloading), when multiple DPUs are stacked together, they can potentially achieve the forwarding capability of network devices. Hence, a possible solution could be to completely abandon switching ASICs and instead use DPUs to offload part of the x86 gateway data plane logic, creating a fast/slow forwarding plane. However, this solution poses two problems for us: First, the original x86 gateway code is not based on fast/slow separation, so offloading to the fast plane requires substantial modifications to the x86 gateway-side code. Second, current DPUs are mainly designed for server-side acceleration, not for native network device use. They typically support large flow tables but may not support LPM forwarding (*e.g.*, BF3) or may conduct software LPM rather than using TCAM (*e.g.*, IPU), which severely degrades the LPM performance of the cloud gateway (such as VXLAN routing lookups) at the traffic convergence point.

Server-based full offloading to DPUs. In fact, some DPUs have started to support P4 programming [7], which led us to consider whether we could offload the entire data plane logic of the Tofino gateway to DPUs, with the x86 gateway CPU only responsible for table entry distribution. The benefit would be maximizing reuse of our Tofino gateway code. However, despite supporting P4 programming, these DPUs still lack sufficient data plane resources to support the demands of cloud-scale gateways, as they were originally designed for server-side network acceleration (*e.g.*, Intel IPU has insufficient LPM resources, which may risk supporting large L3 forwarding tables). While FPGAs could implement the complete Tofino-based gateway data plane, the substantial redevelopment effort would delay deployment, and their performance (e.g., throughput and latency) falls short of Tofino's.

Solutions with packet-level load balancing. Since the bottleneck of our x86 gateway is caused by single-core overload due to RSS, we have also started to focus on packet-level load balancing solutions in the industry, such as Intel DLB [36] and Octeon 10 [14]. The former requires using some cores of a CPU to generate in-order sequence numbers, which occupies valuable CPU cores that could otherwise be used for tenant service processing. The latter uses on-chip hardware resources for traffic spray and in-order preservation, but can only perform packet-level load balancing for its on-chip ARM cores. Since Octeon 10 is typically sold as a DPU card, its power limit (usually 100W) directly restricts the performance of its ARM cores, restricting its usage for high-bandwidth cloud gateways. Currently, the packet-level load balancing capabilities in the market are all integrated on-chip and not exposed via external interfaces, which limits their usage scenarios.

3 Albatross Overview

3.1 Design Goals

Significant performance boost over 1st gen x86 gateways. If Tofino is no longer evolvable and no suitable alternative ASICs are found, we will have to revert to the 1st gen x86 gateway, with only 25Mpps per gateway unit [31]. Clearly, this cannot fill the performance gap left by Tofino. Meanwhile, we observed that the performance of DPUs and CPUs has been growing steadily in recent years. For example, in 2024, DPU reached 400Gbps [7], and by 2025, they are expected to reach 800Gbps [25]. These DPUs can provide up to 1Tbps of I/O when multiple cards are attached to a server via PCIe. As for CPUs, both Intel and AMD have released new CPUs with a large number of cores. For example, AMD's EPYC 9005 series [5] can have up to 192 cores per CPU, or 394 cores with dual-socket deployment. Assuming each core handles 1Mpps, the performance can reach nearly 400Mpps. Although this is still behind that of Tofino, it represents a significant improvement compared to the 1st gen x86 gateways.

Gateway built with commodity hardware preventing vendor lock-in. Based on the above analysis, we are inclined to use the server + DPU solution. However, the unavailability of Tofino has made us more cautious about the supply-chain risks of ASICs. x86 CPUs are off-the-shelf chips that are readily available, while there are many types of DPUs, with different vendors sharing the market, making our choice more prudent. We have paid special attention to DPU shipment volumes, future roadmaps, and supply-chain security. Ultimately, to minimize vendor lock-in risks, we have opted to use FPGA as a replacement for DPUs on Albatross. Unlike the emerging and fragmented DPU market, the FPGA market is mature and stable, dominated by two leading vendors (Xilinx and Altera) in intense competition.

Gateway code reuse as much as possible. After deciding to use the server + FPGA solution, we began the development of the new gateway. As a cloud vendor, we aim to reuse validated code from the previous two generations of gateways, to reduce development effort, minimize bugs, and shorten time-to-market cycle. As discussed earlier, the server + FPGA solution has various implementation choices. For example, if we opt for partial offload to FPGA, the original x86 gateway code needs to be modified into a fast/slow forwarding plane, preventing code reuse. If we opt for full offload to FPGA, since we don't use a P4-based DPU, the FPGA would need to implement the entire forwarding logic of the Tofino gateway. Both options would lead to significant work. Therefore, we adopt a third option, where the FPGA only accelerates data plane logic (*e.g.*, traffic distribution to CPU cores, traffic tagging, reordering, *etc*), while all packet processing remains on the CPU. This way, the CPU can reuse the entire code from the 1st x86 gateway (the P4 code will continue to be maintained for several years until the 2nd gen gateways are fully replaced), and the FPGA only handles partial acceleration, keeping development effort manageable.

Packet-level load balancing to mitigate server CPU overload. Cloud traffic patterns are difficult to predict and can be highly bursty. Therefore, the CPU cores need to tolerate sudden traffic surges. In 1st gen x86 gateways, traffic was distributed by RSS and caused CPU core overloading by heavy hitters. To reduce the overload risk, the gateway had to operate with a low water level, resulting in low resource utilization. To address this, we need to support packet-level load balancing (PLB), dispatching heavy-hitter flows across all CPU cores while maintaining per-flow packet order. Particularly, we implement PLB on FPGA and reserve all the CPU processing capacity to the gateways' services.

Multi-tenant performance isolation to ensure tenant SLA. Albatross must support multi-tenant performance isolation under extreme conditions. A single tenant's traffic burst or abnormal behavior should not affect the traffic processing of other tenants. While the Tofino gateway can absorb abnormal tenants' traffic with its massive pipeline capacity, achieving the same performance isolation capability based on the server and FPGA architecture of Albatross presents challenges.

Containerized gateway for better resource utilization. The previous two generations of gateways were deployed directly with bare-metal hardware and maintained separately by different service teams. Each service team had to individually operate the hardware and consume operational costs of the gateway clusters. The x86 gateway cluster maintained low water level to ease single core

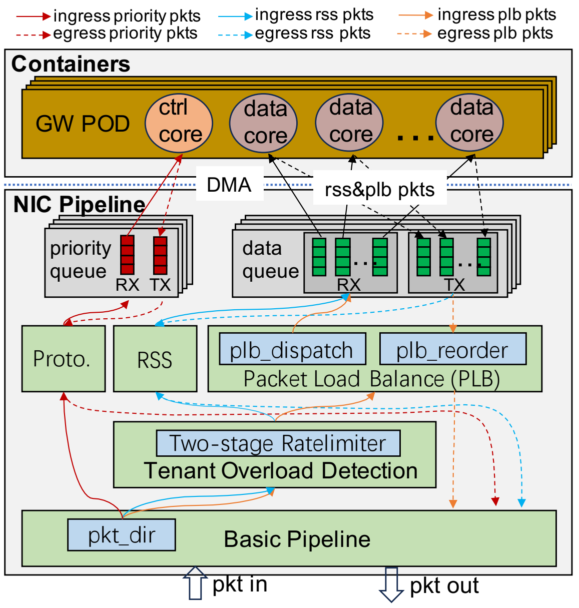

Figure 1: Albatross forwarding plane overview.

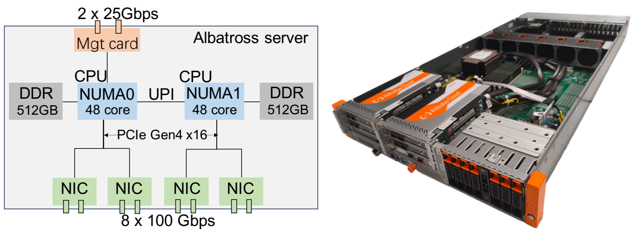

Figure 2: (left) Albatross architecture; (right) Albatross server picture.

overload. The Tofino gateway cluster, despite its high capacities, still required 4-8 gateways for backup and failure recovery, resulting in high setup costs. To reduce such costs through resource sharing, we plan to use containerization to virtualize hardware resources (FPGA and CPU) to support multiple gateway instances.

3.2 System Overview

To achieve the above design goals, we designed the Albatross cloud gateway platform. Albatross consists of two main components: a NIC pipeline implemented on FPGA, and containers hosted on the server CPUs, as shown in Fig. 1.

NIC pipeline. The FPGA-based NIC pipeline implements two functions: inter-core load balancing and gateway overload protection.

For inter-core load balancing, in addition to supporting RSS (*i.e.*, flow-level load balancing), Albatross also implements a packet-level load balancing (PLB) mechanism (see §4.1). In PLB mode, the NIC pipeline sprays ingress traffic at the packet level to the RX data queues, which is then processed by the corresponding CPU core. Since packets from the same flow will be sent to different CPU cores for processing, out-of-order issues will arise in the egress direction. To address this, we have specifically implemented a reorder module in the egress direction to ensure that packets are sent out in order. PLB helps solve CPU core overload issues caused by heavy-hitter flows, incast, and similar scenarios, while also improving the utilization of multi-core CPUs. The challenges of PLB lie in: 1) the complexity of cloud services and the intricate software stack on general-purpose CPUs result in significant delay jitters and severe out-of-order issues; 2) packet drops at the CPU will lead to head-of-line blocking due to unreleased reorder buffer, which will increase the overall packet latency significantly.

To alleviate the overload situation when the entire gateway is saturated due to traffic anomalies or attacks, we specifically designed a gateway overload protection mechanism (see §4.3) to safeguard the entire gateway. The implementation consists of two parts: 1) Tenant overload detection, which is used to detect which tenant causes the gateway to be overwhelmed. The challenge here is to detect the root cause among millions of tenants with limited FPGA resources; 2) Protocol packet prioritization, which allows control plane protocol packets (*e.g.*, BGP packets) to be handled by a dedicated priority queue, ensuring them unaffected even when the entire data plane is saturated.

Fig. 1 shows the flow of packets within the NIC pipeline. Among them, the basic pipeline (elaborated in appendix §A) implements packet classification through pkt_dir when packets enter the ingress NIC pipeline. Specifically, pkt_dir splits the ingress packets into protocol priority packets (priority pkts) and data packets (RSS pkts and PLB pkts). These three types of packets are forwarded through different queues/paths. The pkt_dir is programmable, and the classification of packets into each type can be configured by the containers. Specifically, the containers configure whether packets are delivered to them as whole packets or just as headers. Using header-only delivery can significantly save PCIe bandwidth between the FPGA and CPU, especially when handling large payload packets (*e.g.*, Jumbo frames that have up to 8,500 bytes Ethernet payload [9]). In addition, certain stateful data plane packets are best not processed with PLB, such as Zoonet probe packets [37], service health check packets, and packets for vSwitch to learn cached entries from the gateway [35]. Their low volume is negligible, and scattering with PLB would add unnecessary inter-core consistency maintenance overhead.

Containers. Albatross achieves containerized management and deployment of gateways through Alibaba Cloud's customized K8s components, ACK [8]. By incorporating virtualization techniques, Albatross addresses the issues of low resource utilization and poor scalability of the previous two generations of gateways. In Albatross, the basic unit of container deployment is the gateway pod (GW pod). A single-role gateway can be deployed within a single GW pod. To improve overall system resource utilization, multiple gateways for different services can be deployed as GW pods on the same physical server. Each GW pod implements the functionality of an individual gateway. Therefore, each GW pod has CPU cores responsible for the gateway's data plane (data cores) as well as CPU cores responsible for the gateway's control plane (ctrl cores).

The challenges for containerized gateway lies in: how to realize high density of GW pods. We found that the increasing density of gateway instances significantly raises the number of BGP peering sessions it needs to maintain, putting huge pressure on the the uplink switch's control plane CPU. To prevent control plane CPU saturation, we need to pay special attention to this challenge.

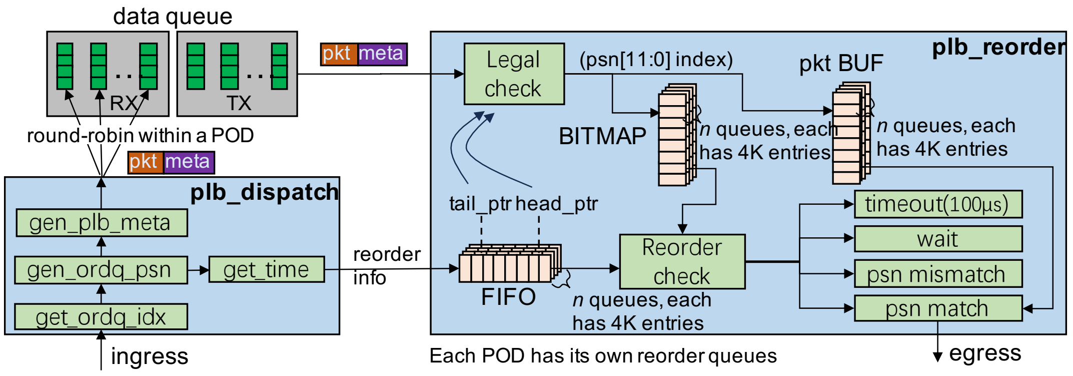

Figure 3: Packet Load Balance (PLB) Pipeline.

Hardware selection and deployment. Albatross has been deployed in Alibaba Cloud for over one year. The current architecture and server picture of Albatross is shown in Fig. 2. Albatross uses a dual-NUMA CPU architecture, with each NUMA containing 48 CPU cores, and the NUMAs are connected via a UPI bus. Each NUMA is connected to 512GB of DDR5 memory. Albatross is configured with a total of 4 2x100Gbps FPGA-based SmartNIC cards, connected to the NUMAs via PCIe Gen4 bus, providing a total I/O throughput of 800Gbps. Two of the NICs are connected to one NUMA, while the other two NICs are connected to the other NUMA. Albatross is also equipped with a 2x25Gbps management NIC, serving as the management port for both the hardware system and the containers. It is worth noting that, to quickly build a production-ready gateway in response to Tofino's discontinuation, we did not use the latest CPU model. Instead, we chose an internally widely used server CPU model for hosting tenant VMs to amortize hardware costs. Nevertheless, we still followed specific criteria for CPU selection, which will be detailed in §7.

4 Albatross Pipeline

4.1 Packet-Level Load Balancing

The PLB primarily contains two modules: plb_dispatch and plb_reorder, responsible for packet spray in the ingress and out-of-order packet reordering in the egress, respectively.

Packet dispatch. In Fig. 3, the plb_dispatch module sprays ingress packets to RX data queues in front of the CPU cores in a round-robin manner. Each GW pod has dedicated CPU cores and data queues, the spraying of packets from different GW pods does not interfere with each other. However, packets of the same flow will be sprayed to different queues and processed by different CPU cores, causing out-of-order delivery at the egress due to varying core processing latencies. To resolve this, PLB maintains multiple order-preserving queues in the egress direction. Before dispatching packets to CPUs, PLB sequentially inserts their arrival order into the order-preserving queues. This information is used to verify the transmission order of packets in the egress by detecting any disorder. To implement the above order-preserving mechanism, plb_dispatch tags a meta header for each packet, which includes a packet sequence number (PSN) to track the arrival order of the packet. Note that the meta header will be sent along with the packet to the CPU and subsequently returned to the NICs for packet reordering. Additionally, plb_dispatch sends reorder info to the order-preserving queue for order verification. The reorder info includes the PSN. Besides, it also contains a timestamp for time-out determination as latency jitters and packet drops may occur during CPU processing. Since we reserve multiple order-preserving queues, the queue for a packet is selected based on its 5-tuple hash (conducted in get_ordq_idx in Fig. 3). The PSN is assigned based on the order within the selected queue.

Packet reorder. The plb_reorder module in Fig. 3 handles packet reordering in the egress using three key data structures: FIFO, BUF, and BITMAP. These structures have the same number of copies (n) and same sizes per copy (4K entries). FIFO: This is the order-preserving queue, where each element is a reorder info structure. For every packet arriving in the ingress, an element is appended to the tail of the FIFO. A packet can be transmitted in order only after it has been processed by the CPU and its corresponding reorder info has reached the FIFO head. The FIFO uses a header_ptr (for dequeue) and a tail_ptr (for enqueue). BUF: A memory buffer that stores packets and their meta headers after processing by the GW pod but before transmission. For header-only delivery, only packet headers and meta headers are stored. BITMAP: A lightweight mirror of BUF that contains only the minimal information necessary (1 valid bit and PSN) for order-preservation checks. It ensures efficient hardware implementation of order comparisons, such as verifying whether the packet at the FIFO head has been processed by the GW pod.

The plb_reorder module has mainly two functions: legal check and reorder check. The purpose of the legal check is to read packets processed by the CPU from the TX data queues, verify their validity, and write them to BUF and BITMAP memory. Validity is determined by quickly checking whether the packet's PSN falls within the range of packets currently queued in the FIFO. Specifically, if the meta.psn[11:0] of the packet falls between header_ptr and tail_ptr, it is considered valid; otherwise, it is deemed invalid. For valid packets, the packet and its meta header are written to BUF memory using psn[11:0] as the memory index, and BITMAP is updated accordingly to indicate that the packet has been written back from the CPU. For invalid packets (essentially timed-out packets), a best-effort strategy is adopted for transmission without reordering. Specifically: If it is a complete packet, it is sent directly. For header-only delivery packets, the NIC buffer is checked to see if the payload is still retained. If the payload is available, the packet is sent; if the payload has already been released, the header is dropped. It is important to note that legal check only examines the lower 12 bits of the PSN. As a result, there may be cases where the packet has already timed out, but its lower 12 bits still fall within the FIFO range. Such packets will pass the legal check but will later be identified during the reorder check. Since the probability of this scenario is low, legal check achieves efficient validation for the vast majority of packets with minimal overhead.

The purpose of the reorder check is to continuously monitor the packet at the head of the FIFO at the FPGA's operating frequency to determine whether it has been processed and sent back by the GW pod. If the packet has been returned, it is transmitted; otherwise, the check continues until a timeout occurs. There are four possible cases for reorder check. Case 1: If the FIFO head element has been queued for over $100\mu s$, it is directly released. According to our statistics, the processing latency for most cloud gateway services is less than $50\mu s$. Case 2: If the GW pod has not yet processed the packet (*i.e.*, the valid bit in the BITMAP is 0), the system continues to busy-wait. Case 3: If the GW pod has returned the packet, but the PSN does not match, it indicates a timed-out packet was sent back and happened to pass the legal check. In this case, the packet is still sent using a best-effort approach. Case 4: If the GW pod has returned the packet and the PSN matches, the packet is transmitted in order.

Reorder queue granularity. A key question worth discussing is: at what granularity should reordering be performed? A naive approach is to perform reordering at a per-flow granularity. However, a cloud gateway typically needs to handle millions of concurrent flows. Using per-flow reordering would require allocating a separate reorder queue for each flow, posing significant challenges in terms of dynamic buffer management. Albatross instead performs reordering for a group of flows, which significantly reduces the number of reorder queues and avoids frequent buffer allocation and release. Specifically, Albatross maintains 1-8 reorder queues per GW pod, which each queue contains 4K entries. Generally, pods with more CPU cores are allocated more reorder queues. Under fixed resource constraints, Albatross's design strikes a balance between two extremes: C1: A larger number of reorder queues reduces the heavy-hitter size that a pod can tolerate. C2: A smaller number of reorder queues increases the risks of head-of-line (HOL) blocking.

Here is a discussion of C1 (C2 discussed later). Assume the FPGA's buffer size for reorder queues is a constant, increasing the number of reorder queues reduces the queue length. This also decreases the maximum packets-per-second (pps) that a reorder queue can handle, if the maximum processing latency for a packet is fixed. That is to say, the presence of a heavy-hitter flow with high pps will lead to severe packet loss. In Albatross's implementation, the queue length is set to 4K, ensuring it can buffer $100\mu s$ packets at 40Mpps.

Head-of-line (HOL) issues and handling. Since the reorder queue is a FIFO queue, if the packet at the head of the queue is not processed, it will block the transmission of subsequent packets, causing a head-of-line (HOL) blocking issue. Minor HOL issues are tolerable with the time-out mechanism, but if HOL occurs for a long time, it will significantly increase the forwarding latency of subsequent packets and may even cause the BUF queue to become full, leading to packet loss. It is important to note that since the NIC pipeline currently ensures order for processing by external CPU cores, there are many external factors that could cause HOL, such as slow CPU processing, rate-limiting packet loss, ACL packet loss, RX/TX queue congestion, and driver-induced packet loss. Albatross has invested considerable effort into addressing HOL issues. The following are the technical details.

1) Using multiple order-preserving queues. Clearly, the HOL in one order-preserving queue will not affect other queues, which reduces the overall risk of HOL. We allocate multiple reorder queues to each GW pod, and the number of reorder queues used by a pod is proportional to the number of CPU cores. This ensures that the number of heavy-hitter flows each reorder queue can tolerate is consistent.

2) Setting a drop flag for the GW pod. Cloud gateways handle traffic subject to rate-limiting and ACL rules, and packet drop occurs when these rules are triggered. To prevent HOL, we add a drop flag in the meta header. When the GW pod triggers such packet drops, it actively sets this flag to notify the NIC pipeline to drop the packet. At this point, the NIC pipeline actively releases the reorder resources, including FIFO, BUF, and BITMAP.

3) Ensuring that the cloud gateway's processing latency is less than $50\mu s$. Generally, we assume that the processing latency for major cloud network services is less than $50\mu s$. However, during the adaptation of Albatross, we identified several corner case code branches. Due to code quality issues, these branches caused forwarding latency to become excessively high, even reaching the millisecond level. We have already fixed these code branches.

4) System and driver optimization. We identified and resolved issues that caused abnormal latency increase at the hardware and driver levels. For example, we discovered and fixed problems caused by enabling Automatic NUMA Balancing, insufficient PCIE driver descriptors, and a too-small DPDK_RTE_MEMPOOL_CACHE, which led to the abnormal increase in latency.

5) PLB fallback to RSS. If the previous methods do not work and we are unable to pinpoint the root cause, the GW pod can dynamically switch from PLB mode to RSS mode to attempt remediation. This method has not been triggered in the current online deployment.

4.2 PLB Performance Optimization

Intuitively, PLB would seem to utilize cache locality less efficiently than RSS, which raises concerns that cloud gateways based on PLB might exhibit worse overall performance compared to first-generation x86 gateways based on RSS. However, our tests show otherwise. We first tested the performance of RSS and PLB under a typical cloud gateway workload, VPC-Internet (definition in Tab. 2), with 500K concurrent flows, as shown in Fig. 4. The results indicate that, under 1, 20, and 40 CPU cores, the per-core performance of PLB and RSS for the cloud gateway workload is nearly identical, with a difference of less than 1%. Other workloads on the cloud gateway exhibit similar results. This outcome even exceeded our expectations. Through analysis, we found that this is due to the characteristics of cloud gateway workloads. Cloud gateways handle a massive number of tenants, each with different network configurations, leading to an enormous number of table entries. Moreover,

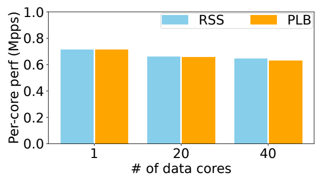

Figure 4: Perf comparison of VPC-Internet service.

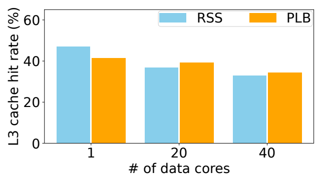

Figure 5: L3 cache hit comparison.

the complexity of the workloads results in long table entries, often hundreds of bytes. Even for a single workload gateway, multiple cascading table entries are typically involved. According to statistics, table entries in a typical cloud gateway occupy several GB of memory, far exceeding the approximately 200 MB of CPU cache available. The large number of concurrent tenant flows causes frequent cache replacements during table lookups, resulting in a low L3 cache hit rate of about 30%-45%, as shown in Fig. 5. Additionally, since L3 cache is shared across cores, both RSS (flow-based hashing) and PLB (packet-based spraying) ultimately achieve similar performance.

The performance test results of PLB have a significant impact on the model selection of Albatross. VPC-Internet is a highly representative workload, with an L3 cache hit rate of around 35%, indicating that our gateway services involve a large amount of memory read and write operations during forwarding. Therefore, in addition to considering the CPU cache size for Albatross model selection, we also place great emphasis on memory performance. For example, we prefer models with low memory access latency and high memory frequency. According to our tests, when the memory frequency is increased from 4800 MHz to 5600 MHz (via BIOS modifications), the gateway performance improves by approximately 8%. Another optimization for gateway performance involves NUMA design, where we enhance performance by avoiding cross-NUMA scheduling (see §7). In addition to these, we have also optimized aspects such as CPU Turbo Boost, DDIO, LLC Prefetch, and Hyper-Threading.

4.3 Gateway Overload Protection

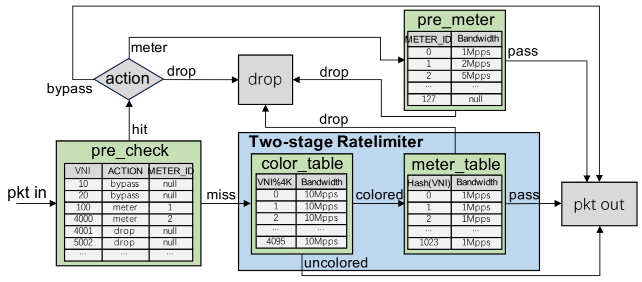

Figure 6: Tenant overload rate-limiting

Essentially, Albatross is still a software-based gateway, and its forwarding performance lags behind Tofino, making it vulnerable to heavy-hitter flows. Under extreme conditions, Albatross may encounter cases such as CPU overload, VF overload, and NIC port overload, which severely degrade overall forwarding performance and violate tenant network SLAs. For instance, a traffic surge from a single tenant, combined with PLB, can overload all CPU cores, causing packet loss for other tenants due to insufficient CPU resources and breaking SLA guarantees for performance isolation. To mitigate the impact of heavy-hitter flows, we employ two gateway overload protection (GOP) techniques.

Two-stage tenant overload rate-limiting. The first GOP technique detects which tenants are causing overload through the NIC pipeline and rate-limits those tenants before their traffic reaches the CPU. Based on our years of experience in cloud gateway operations, most CPU overloads are caused by sudden bursts or anomalies from one or a few dominant tenants. Therefore, the NIC pipeline must identify these dominant tenants. However, with hundreds of thousands of concurrent tenants in a cloud network, assigning rate-limiting meter resources for each tenant on an FPGA would require over 200 MB of SRAM for 1 million tenants, far exceeding the available on-chip memory after provisioning resources for PLB and other capabilities. To address this, we propose a two-stage rate-limiting scheme to reduce resource consumption. Specifically, in the first stage (color_table), we allocate a rate-limiting table with 4K entries for all tenants. Incoming traffic is indexed into the color_table using VNI%4K and subjected to coarse-grained rate limiting, where VNI is the identifier of a tenant. If the traffic exceeds the preset limit, the excess is marked and sent to the second stage (meter_table) for fine-grained rate limiting. In this stage, the marked traffic is hashed based on VNI and rate-limited in the corresponding entry of the meter_table. This two-stage rate-limiting scheme reduces on-chip SRAM usage for 1 million tenants to just 2 MB (100× reduction from the naive approach).

It is worth noting that the second-stage meter_table is implemented as a hash table. Hash collisions may cause some innocent tenants to be incorrectly rate-limited. For example, the overflow traffic from the first-stage color_table might include innocent tenants, and these tenants could hash to the same entry that a dominant tenant occupied in the second stage (If two dominant tenants collide, rate-limiting them does not pose any issues). To address this, we add a pre_check table before the two-stage rate limiter. This table identifies heavy hitters detected in the meter_table with sampling (heavy hitters are more likely to be sampled due to their higher packet rates), then enabling early rate-limiting of heavy hitters in the pre_meter table. This prevents dominant tenants from affecting others in the meter_table due to hash collisions. Since the hash collisions happen at a low probability and the early rate-limiting of heavy hitters takes effect quickly (in one second), the impact of incorrect rate-limiting of innocent tenants is acceptable in production. Since Albatross was deployed, there have been no tenant complaints about incorrect rate-limiting issues. To further reduce the probability of innocent tenants' rate-limiting in the future, we plan to utilize the CPU to detect heavy hitters in advance and then install them to the pre_check and pre_meter table for avoiding triggering hash collisions in the meter_table. Both pre_check and pre_meter have only 128 entries, consuming very limited resources. Additionally, certain top-tier customers who do not want to be rate-limited can be configured in the pre_check table to bypass all rate-limiting logic.

High priority assignment for protocol packets. The second GOP technique assigns higher priority to protocol packets and routes them through dedicated RX/TX priority queues. This approach is taken because packet loss for protocol packets can directly impact the operational status of all containers on the gateway. For instance, GW pods on Albatross use BGP and BFD protocols to

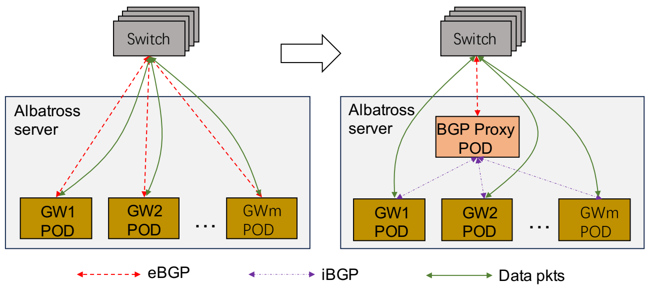

Figure 7: (left) Original BGP scheme; (right) BGP proxy scheme. Each GW pod establishes BGP peers with uplink switch.

maintain connections with upstream switches. BFD [24] is a fast link failure detection protocol that can accelerate the convergence speed of BGP. Typically, losing three consecutive BFD probe packets is enough to trigger a link failure detection and disable the entire link. Therefore, in scenarios where Albatross experiences traffic overload, even a few lost BFD packets can result in a link failure being detected, causing BGP to register a neighbor link failure. In Albatross, protocol packets can be configured in pkt_dir to use priority queues.

5 Albatross Container

Albatross implements container support in the NIC pipeline, enabling the concurrent operation of multiple gateways on a single server to improve overall server utilization.

NIC virtualization to support multiple gateway containers. As multiple GW pods share the same NIC pipeline, the NIC resources need to be partitioned to allow each GW pod has its own NIC resources. We primarily adopt the following approaches to achieve NIC virtualization: Splitting NIC queue resources and table entries and allocating them to different containers, thereby enabling the virtualization of NIC network functions. For example, Albatross partitions reorder queues and packet direction tables, assigning portions of these resources to different GW pods. This way, each GW pod exclusively has its own NIC resources and remains unaware of the presence of other GW pods. The resources allocated to each GW pod are typically proportional to its capacities. For instance, a 40-core GW pod is assigned twice as many reorder queues as a 20-core GW pod. To ensure high performance and minimize the overhead introduced by virtualization, we employ hardware-based virtualization through SR-IOV (As shown in Fig. B.1 in appendix §B). Specifically, we virtualize multiple VFs on the NIC's PF, assigning them to different containers. To ensure robustness, each GW pod uses 4 VFs (As shown in Fig. B.2 in appendix §B) and allocates n RX/TX queue pairs per VF, where n equals to the number of data cores of the GW pod.

Increasing container density with BGP proxy. When developing containerized gateways, we encountered a significant challenge: the density of containers was constrained by the BGP peer capacity of the uplink switch (in our gateway-centric architecture, multiple gateways are attached to a group of switches and traffic is routed to the gateways via these switches). Since cloud gateways need to advertise VIP routes externally, each gateway must establish a BGP peer with the switch. However, with containerized gateways, the number of BGP peers that each physical server needs to establish increases significantly. We discovered that the bottleneck for the density of containerized gateways (*i.e.*, the number of GW pods a single physical server can host) lies in the number of BGP peers supported by the switch. Currently, the safe threshold for the maximum number of BGP peers supported by the switch is 64. Exceeding this threshold can lead to slow route convergence in abnormal situations (*e.g.*, switch restarts, unexpected power outages, or hardware failures), requiring up to tens of minutes for route convergence. Given that the maximum number of interfaces on a switch can connect to 32 Albatross servers, under the 64 BGP peer limitation, each Albatross server can host at most two containerized gateways. To overcome this limitation, we developed a BGP proxy solution, as illustrated in Fig. 7. Unlike the original approach, where each GW pod directly established an eBGP connection with the switch, the BGP proxy solution introduces a BGP proxy pod that handles the eBGP connection with the switch. All GW pods on the server establish iBGP connections with the BGP proxy pod instead. This approach reduces the number of BGP peers established with the switch to $1/m$, where m is the number of GW pods on the server. For deployment, we adopt a dual BGP proxy setup per server to enhance robustness.

6 Evaluation

Table 2: Four typical cloud gateway services.

#	GW Services	Explanation
1	VPC-VPC	A VM accesses another VM in the same VPC.
2	VPC-Internet	A VM in VPC accesses Internet.
3	VPC-IDC	A VM in VPC accesses one customer's private data center by hybrid cloud.
4	VPC-CloudService	A VM in VPC accesses cloud services provided by cloud vendors (*e.g.*, log stores, databases, *etc*).

Table 3: Albatross's performance evaluation with different gateway services.

GW service	VPC-VPC	VPC-Internet	VPC-IDC	VPC-CloudService
Packet Rate	128.8Mpps	81.6Mpps	119.4Mpps	126.3Mpps

Table 4: NIC pipeline latency measurement.

Module	RX (μs)	TX (μs)
Basic Pipeline	0.58	0.84
Overload Det.	0.10	0
PLB	0.05	0.35
DMA	3.17	2.98
Sum	3.90	4.17

Table 5: NIC pipeline resource consumption.

Module	LUT	BRAM
Basic Pipeline	42.9%	38.2%
Overload Det.	2.0%	0%
PLB	12.6%	5.0%
DMA	2.5%	1.3%
Sum	60.0%	44.5%

Overall performance and overhead. We evaluate the overall forwarding performance by testing four typical gateway services (listed in Tab. 2) on one Albatross server. For each gateway service, we allocate two 46-core GW pods. Each pod is within a NUMA node, configured 44 data cores and 2 ctrl cores. We generate and send 500K flows (with 256B packets) for each GW pod for all the performance tests. As shown in Tab. 3, the overall throughput of Albatross is around 120Mpps. Since the four gateway services have different processing code and forwarding tables, they experience different packet forwarding rates. The VPC-Internet performance degradation (81.6Mpps) comes from the significant longer processing code and more lookup tables than other gateway services.

Table 6: Albatross's comparison with our 2nd gen gateway (Sailfish). Albatross* is the evolution of Albatross on our roadmap.

Gateway	# of LPM rules	Elasticity	Price/device	Price/AZ	Throughput	Packet Rate	Latency
Sailfish	0.2 M	days	1x	32x	3200 Gbps	1800 Mpps	2 μs
Albatross	>10 M	10 seconds	2x	16x	800 Gbps	~120 Mpps	20 μs
Albatross*	>10 M	10 seconds	2.4x	9.6x	3200 Gbps	~480 Mpps	20 μs

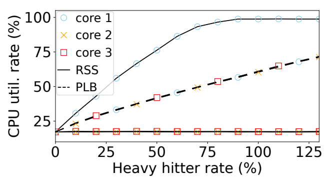

Figure 8: Load balancing comparison.

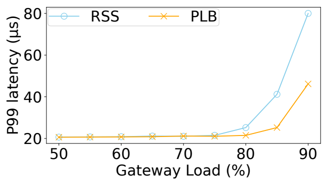

Figure 9: P99 latency comparison.

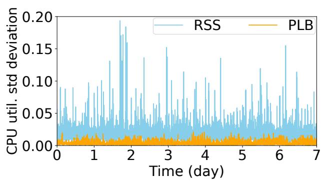

Figure 10: Multi-core util. rate comparison.

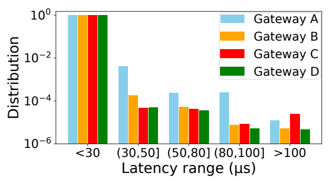

Figure 11: PLB latency distribution in production.

We also evaluate the latency and resource consumption of the FPGA NIC pipeline, shown in Tab. 4 and Tab. 5 respectively. We can see that the overall RX+TX latency introduced by NIC pipeline is around 8 μs, most consumed by DMA procedure. The extra latency introduced by PLB and overload detection is 0.5 μs, which is only a small part of the NIC. Each FPGA contains 912,800 LUTs and 265 Mbits BRAM. The resource consumed by PLB and overload detection is 14.6% LUT and 5% BRAM. While basic pipeline implements major traditional NIC functions (*e.g.*, parser, deparser, *etc*) and a payload buffer (which stores packet payload when using header-payload-split mode), it consumes the majority of LUT and BRAM resources on FPGA.

Comparison with 2nd gen Tofino-based gateway (Sailfish). We conduct a head-to-head comparison of Albatross to Sailfish [31], shown in Tab. 6. The main advantages brought by Albatross are threefold. First, Albatross can accommodate extra-large forwarding tables as it can use hundreds of GB DRAM memory. For example, it can hold >10M LPM rules (for VXLAN routing table) while Sailfish can only hold about 0.2M. Second, Albatross has enhanced elasticity and can prepare a GW pod in 10 seconds while Sailfish's elasticity is tens of days as it must prepare physical clusters. Third, although Albatross has a higher per-device cost (2x), the total cost of setting a new available zone (AZ) has been halved due to Albatross's containerization support (discussed at the end of this section). Compared to Sailfish, Albatross has a significant forwarding performance regression in throughput (by 4x), packet rate (by 18x), and latency (by 10x). This is a predictable degradation since Sailfish employs a programmable switch architecture that can process packets at line rate while Albatross employs a CPU-based architecture that is short of processing performance. Note that the packet rate regression (by 18x) is worse than the throughput regression (by 4x), this is because Albatross has bottleneck at CPU that makes it impossible to achieve line rate with small-size packets.

Readers may have concerns about Albatross's forwarding performance regression. We would like to respond it from two aspects. First, cloud gateway clusters can be classified into two categories, *i.e.*, throughput-sensitive (*e.g.*, > 100Gbps) and throughput-insensitive (*e.g.*, < 100Gbps). According to our statistics, 91% gateway clusters of Alibaba Cloud are throughput-insensitive, which is the major demand that Albatross aims to meet. Second, the 9% throughput-sensitive gateways can be met by the evolution of Albatross. On our roadmap, the next gen of Albatross, denoted as *Albatross** (with 20% increased per-device cost), would improve the throughput to 3200Gbps and packet rate to 400Mpps by employing more powerful latest FPGAs and CPUs in the market. In a word, the primary goal of Albatross is to rapidly validate a new alternative gateway platform and meet the majority of actual demand, leaving the subsequent evolution (Albatross*) to meet the left throughput-sensitive demand.

PLB performance. Compared to RSS, PLB has the advantage of mitigating CPU single-core overload caused by heavy hitters. As shown in Fig. 8, in a scenario with 500K background flows and three forwarding cores with 10% single-core utilization, increasing a heavy hitter's rate from 0 to 130% (relative to the maximum throughput of a single core), RSS can only hash this heavy hitter to one core, resulting in core 1 overload and significant packet loss. In contrast, PLB evenly distributing the heavy hitter across three cores, thus avoiding the single-core performance bottleneck. In real deployment scenarios with a larger number of forwarding cores (*e.g.*, 40 cores), the tolerance of heavy hitters can be enhanced by tens of times. The improved multi-core load balancing by PLB also enhances gateways' tail latency. As shown in Fig. 9, the P99 latency of PLB outperforms RSS when the gateway load exceeds 75%. There is no significant difference in latency when the gateway load is below 75%, since we simulate real cloud network's micro-burst traffic that makes the gateway processes packets without any burden when gateway load is not high.

PLB performance in production deployment. Since Albatross has been deployed in production, we collect operational data to evaluate the effectiveness of PLB. First, we evaluate the multi-core load-balancing effectiveness. We collect two production gateways' CPU utilization in one week and calculate the standard deviation among cores. The two gateways adopt PLB and RSS respectively, each with around 20% load. As shown in Fig. 10, the standard deviation of RSS fluctuates significantly and is much higher than that of PLB. Based on our observation, this is because cloud gateways experience numerous micro-bursts, which can increase the utilization of a single core by about 50% under RSS in less than one second. Such micro-bursts, when spread to tens of cores by PLB, have a negligible impact on the utilization of each core.

Second, we evaluate the latency behavior and head-of-line (HoL) handling. We choose four Albatross gateway pods, A (20% load), B (17% load), C (6% load), and D (5% load), and analyze their packet processing latency distributions, as shown in Fig. 11. It indicates that over 99% of packet processing latencies are less than 30μs, with higher latency's distribution decreasing exponentially. Additionally,

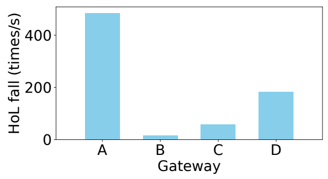

Figure 12: HoL optimization with active drop flag.

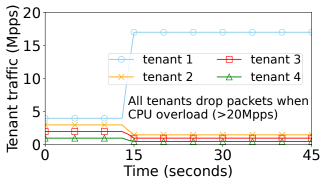

Figure 13: Without tenant overload rate-limiting.

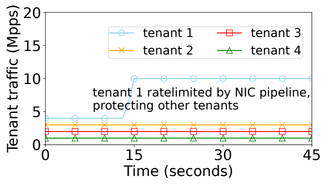

Figure 14: With tenant overload rate-limiting.

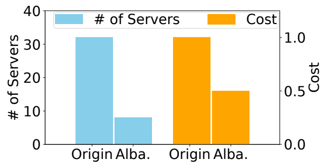

Figure 15: Gateway construction cost comparison

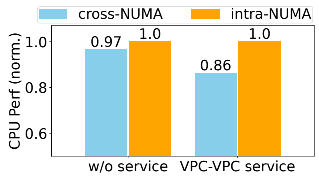

Figure 16: Cross/intra NUMA comparison.

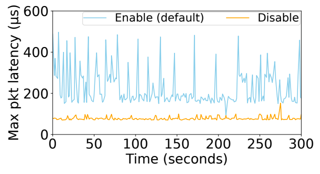

Figure 17: Impact of NUMA balancing.

gateways with higher loads tend to increase the proportion of latencies in the 30-100μs range. Given that the PLB timeout is set to 100μs, packets with latencies exceeding 100μs may experience disordering. Fig. 11 shows that the disordering rates are around 10^{-5} for all four gateways. The overall low latency behavior of Albatross comes from various HoL optimizations. For example, when packet loss occurs on the CPU (e.g., ACL blocking), we use an active drop flag in PLB meta header to notify the NIC pipeline to release the reorder resources occupied by the dropped packets. As shown in Fig. 12, the active drop flag mechanism reduces the occurrence of HoL by several dozen to hundreds of times per second.

Tenant overload rate-limiting performance. We conduct two sets of comparative experiments. The experimental setup includes four tenants with initial rates of 4, 3, 2, and 1 Mpps (total 10Mpps), respectively. The first tenant (*i.e.*, dominant tenant) increases its rate to 34 Mpps at the 15th second, while the other three tenants maintain their rates (total 40Mpps). The four tenants' traffic is sent to an Albatross GW pod that adopts PLB and has a maximum throughput of 20 Mpps. The tenant overload meters are set 10Mpps, with first stage set 8 Mpps and second stage set 2 Mpps. As shown in Fig. 13, without tenant overload rate-limiting, the total traffic of the four tenants (40 Mpps) at the 15th second exceeds the overall throughput (20 Mpps), causing the CPU to indiscriminately drop packets, leading to a 50% packet loss for all tenants. This means that a dominant tenant's traffic burst affects other tenants' SLAs. In contrast, with tenant overload rate-limiting, as shown in Fig. 14, the traffic of tenant 1 is directly rate-limited to 10 Mpps in the NIC pipeline, resulting in a total of 16 Mpps CPU throughput, which does not exceed the overall throughput. Therefore, the other tenants are not affected. According to our design, if a dominant tenant and innocent tenants collide in the second-stage meter_table, leading to incorrect rate-limiting, we can resolve this issue within a few seconds by migrating the dominant tenant to the pre_meter for early rate-limiting.

Benefits of cloud gateway containerization. One significant benefit of Albatross is the cost reduction by containerization, especially for smaller gateway clusters. For instance, when setting up a new available zone, the cloud network team need to build eight types of gateway clusters (XGW, IGW, VGW, *etc.*), with four gateways per cluster. As shown in Fig. 15, in the 1st and 2nd gen gateway forms, setting up a new available zone (AZ) requires deploying 32 physical gateways. With Albatross, these 32 gateways can be distributed across 8 Albatross servers, with each server hosting 4 GW pods. This reduces the number of servers by 75%. Since the cost of an Albatross server is twice that of the previous two generations, the total cost is reduced by 50%. Due to the power consumption of the 1st-, 2nd-, and 3rd-gen single gateway being 500W, 300W, and 900W, respectively, the total power consumption for the eight gateway clusters is 12,000W (three 1st-gen clusters, five 2nd-gen clusters), while the total power consumption for 3rd-gen gateway clusters is 7,200W, which is reduced by 40%.

7 Lessons learned and experiences

NUMA architecture affects Albatross's performance. Fig. 16 illustrates the gateways' performance comparison between whether allocating CPU and memory across NUMA nodes (cross-NUMA) or inside a NUMA node (intra-NUMA). The results show that cross-NUMA degrades the VPC-VPC service's performance by 14%, and degrades the performance by 3% without any network service. This degradation is caused by increased latency in memory allocation and access, unnecessary overhead in maintaining cache coherence, and delays introduced by cross-NUMA scheduling. Therefore, Albatross requests gateway pods operate within a NUMA node. However, after we restrict the gateway pod inside a NUMA node, we still observed unexpected cross-NUMA scheduling under heavy traffic conditions. Fig. 17 shows that there are latency bursts under 90% load when a gateway pod is restricted to a NUMA node. This is because typical operating system kernels have the numa_balancing feature enabled by default, which tries to automatically move tasks and application data closer to the memory they are accessing [23]. However, since we restrict the gateway pod to a NUMA node, Albatross could not benefit from this feature. By disabling numa_balancing, we significantly reduce the maximum packet latency and the latency jitter.

Performance optimization with PLB meta header. We explored two alternative strategies that where to attach PLB meta header: attached at 1) packet head, or 2) packet tail. For the first strategy, naively inserting the PLB meta to the packet head room will affect the extensive packet encapsulation and decapsulation. If alternatively placing the PLB meta in the private room of the rte_mbuf within the DPDK driver, it will introduce extra data copying overhead, which can degrade forwarding performance by 33.6%. Conversely, for the second strategy, placing PLB meta at the packet tail does not introduce these issues, as cloud gateways will not process packet tails. For the cases that gateway pods have to set flags in PLB meta, it will not significantly affect the overall performance since such cases' frequencies are relatively low. Therefore, we select the second strategy as our final solution.

The migration of existing cloud gateways to Albatross. To ensure the stability of cloud gateways during migration, we plan to migrate the gateways according to the following priority sequence: 1) newly established gateway clusters; 2) out-of-warranty gateway clusters; 3) light-load gateway clusters; 4) heavy-load gateway clusters. Each gateway cluster must undergo staged gray testing prior to production deployment. Additionally, to ensure high availability, some core cloud gateways are designed to deploy both container-based and traditional physical forms during the migration, serving as mutual primary and backup gateways.

Leveraging container elasticity to enhance the stability of cloud gateways. We will build redundant Albatross clusters in advance. Facing the unexpected growing load that approaching the throughput capacity, Albatross can leverage the container elasticity (10 seconds) to quickly prepare a new GW pod (with more CPU cores and reorder queues) and then migrate the traffic to the new GW pod in case of violated network SLAs. In order to guarantee no service disruption, before the original GW pod withdraw the BGP route, the new GW pod has to advertise the BGP route first and validate packets are processed normally for a while (*e.g.*, 30 seconds). Compared to traditional physical gateway forms, which usually requires tens of days to prepare a new gateway cluster, Albatross' elasticity can be used to enhance the capability to handle traffic bursts and improve the overall system stability.

Stateful network function (NF) support with PLB. Apparently, Albatross can support stateful NFs with RSS mode. Compared to stateless NFs, the stateful NFs (*e.g.*, SNAT and Layer-4 load balancer) need to maintain and update flow states (or sessions) for correct forwarding. Using PLB mode, the multi-core write operation to the same flow state brings additional state synchronization overhead. Based on our experience, whether such state synchronization overhead has a significant impact on forwarding performance depends on whether a stateful NF is write-heavy (*i.e.*, each flow has to write flow states frequently, especially per-packet write operation) or write-light (the opposite of write-heavy). If one stateful NF is write-light (*e.g.*, establishment and termination of sessions), we found that the NF's performance scales (approximately) linearly with the number of CPU cores, which is very promising. However, if one stateful NF is write-heavy (*e.g.*, per-session counters), we found that the lock contention for per-flow state write operation leads to significant performance degradation, and the more CPU cores we use, the worse of overall performance. We also found that even if we remove the locks for per-flow state write operation, such overall performance degradation remains largely unchanged. The reason is that the multi-core cache coherence mechanism leads to significant cache-misses and cache-references. Therefore, the most important optimization for write-heavy stateful NFs is minimizing locks used and multi-core cache contention. For example, we can: 1) transform shared-states into local-states (*e.g.*, per-core states), and 2) partition CPU cores into groups and spray packets across a subset of cores..

Future FPGA offloading plan. We reserve sufficient room of FPGA usage for future evolution since the cloud gateways must continuously develop new features to meet growing business demands. In our current plan, the following three categories of operations will be offloaded to the FPGAs. 1) *Session offloading*. As discussed above, the write-heavy stateful NFs (session-based) with PLB mode experience significant performance degradation due to state synchronization. However, if these stateful NFs use RSS mode, they cannot benefit from multi-core load balancing and suffer from single-core heavy-hitter impact significantly. Therefore, we plan to offload the sessions to FPGAs to improve Albatross's ability to handle stateful NFs; 2) *Computation-intensive operations.* Since CPU is usually the bottleneck of Albatross's processing capability, we plan to offload some operations to save the CPU power, *e.g.*, encryption/decryption, traffic counting/metering, *etc*; 3) *Operational functions.* We also plan to offload some tenant-transparent functions to FPGAs, *e.g.*, billing, firewalls, *etc.*

8 Related Work

Different cloud vendors have developed distinct network virtualization architectures. For example, Azure uses an architecture based on distributed vSwitches [17] and offloads processing to SmartNICs [18]. Overflow traffic on SmartNICs is handled by a remote DPU resource pool [2]. In Andromeda, Google employs Hoverboard to cache the long tail of low-bandwidth flows, reducing the overhead of controller reconfigurations [16]. Alibaba Cloud adopts a centralized cloud gateway to handle east-west and north-south traffic forwarding [35]. Initially, x86 server clusters were used to implement the cloud gateway, but later it transitioned to Tofino [31], which offers high performance and cost-effectiveness. However, with the discontinuation of Tofino's evolution, we are pushed to update our technical roadmap.

When falling back from Tofino to x86 CPUs, the challenge is the single-core bottleneck caused by RSS [20]. Packet-level load balancing (PLB) is a common industry solution. For example, early network processors dispersed packets within the chip and reordered them using hardware logic [1, 21]. Since processing was confined to the chip and L3 forwarding logic was relatively simple, latency jitters were relatively low, and reorder buffer design was straightforward. In contrast, Albatross's reorder buffer reserves up to 100μs of queuing delay to accommodate the latency jitter caused by complex cloud network workloads on CPUs. Intel's latest CPUs include DLB capabilities, but implementing DLB consumes additional CPU cores for tasks like per-packet sequence tagging [36]. Albatross offloads PLB entirely to FPGA, preserving CPU resources for tenant workloads. DPUs like Octeon 10 also support PLB, but only for their internal ARM cores [14]. Exposing PLB capabilities to external CPUs via PCIe could unlock significant performance potential but introduces greater challenges in exception handling. In Albatross, when a CPU core experiences an exception or packet loss, explicit notifications to the FPGA are required for reorder resource release.

On previous x86 gateways, we designed several methods to prevent overload from heavy-hitter flows. CloudSentry [27] uses CPU water level to trigger data-plane sampling, enabling lightweight heavy-hitter detection and rate limiting, but this process takes tens of seconds. MIMIC [34] leverages SmartNICs for real-time heavy-hitter detection in the data plane, significantly reducing rate-limiting time, but it requires maintaining large concurrent flow tables. Albatross uses a two-level rate-limiting table on FPGA, balancing heavy-hitter detection time and memory overhead.

9 Conclusion

Albatross is Alibaba Cloud's 3rd gen gateway in response to the unexpected cessation of Tofino's evolution. It is developed using off-the-shelf x86 CPUs and FPGAs. Specifically, to address the CPU overload issue, FPGA is used for implementing packet-level load balancing to CPU cores and a two-level rate limiter. To improve the gateway's resource utilization, multiple gateways are deployed as containers on the same Albatross server.

This work does not raise ethical issues.

10 Acknowledgment

The authors would like to thank the shepherd David Wetherall and the anonymous reviewers for their constructive comments.

References

[1] J. R. Allen, B. M. Bass, C. Basso, R. H. Boivie, J. L. Calvignac, G. T. Davis, L. Frelechoux, M. Heddes, A. Herkersdorf, A. Kind, J. F. Logan, M. Peyravian, M. A. Rinaldi, R. K. Sabhikhi, M. S. Siegel, and M. Waldvogel. 2003. IBM PowerNP network processor: Hardware, software, and applications. *IBM Journal of Research and Development* 47, 2.3 (2003), 177–193.

[2] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, Balakrishnan Raman, Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, and Srikanth Kandula. 2023. Disaggregating Stateful Network Functions. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1469–1487. https://www.usenix.org/conference/nsdi23/presentation/bansal

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (July 2014), 87–95.

[4] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 99–110.

[5] AMD Corporation. 2025. 5th Generation AMD EPYC Processors. https://www.amd.com/en/products/processors/server/epyc/9005-series.html. (Accessed: 2025-07-14).

[6] AMD Corporation. 2025. AMD EPYC 9965. https://www.amd.com/en/products/processors/server/epyc/9005-series/amd-epyc-9965.html. (Accessed: 2025-07-14).

[7] AMD Corporation. 2025. AMD Pensando DPU Technology. https://www.amd.com/en/products/data-processing-units/pensando.html. (Accessed: 2025-07-14).

[8] Alibaba Cloud Corporation. 2025. Container Service for Kubernetes (ACK). https://www.alibabacloud.com/en/product/kubernetes?_p_lc=1&spm=a3c0i.187803.6791778070.166.14376b64abnz4j. (Accessed: 2025-07-14).

[9] Alibaba Cloud Corporation. 2025. Jumbo Frames. https://www.alibabacloud.com/help/en/ecs/user-guide/jumbo-frame/. (Accessed: 2025-07-14).

[10] Broadcom Corporation. 2025. Trident4/BCM56880 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series. (Accessed: 2025-07-14).

[11] Cisco Corporation. 2025. Cisco Silicon One Q200 and Q200L Processors Data Sheet. https://www.cisco.com/c/en/us/solutions/collateral/silicon-one/datasheet-c78-744312.html. (Accessed: 2025-07-14).

[12] Intel Corporation. 2025. Intel Infrastructure Processing Unit (Intel IPU). https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html. (Accessed: 2025-07-14).

[13] Intel Corporation. 2025. Intel Tofino. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html. (Accessed: 2025-07-14).

[14] Marvell Corporation. 2023. Marvell OCTEON 10 DPU Platform. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-octeon-10-dpu-platform-product-brief.pdf. (Accessed: 2025-07-14).

[15] NVIDIA Corporation. 2025. NVIDIA BlueField Networking Platform. https://www.nvidia.com/en-us/networking/products/data-processing-unit/. (Accessed: 2025-07-14).

[16] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 373–387. https://www.usenix.org/conference/nsdi18/presentation/dalton

[17] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 315–328. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/firestone

[18] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

[19] Linux Foundation and DPDK. 2010. Data Plane Development Kit (DPDK). https://www.dpdk.org/. (Accessed: 2025-07-14).

[20] Stephen D Goglin and Linden Cornett. 2009. Flexible and extensible receive side scaling. US Patent 7,584,286.

[21] S. Govind, R. Govindarajan, and Joy Kuri. 2007. Packet Reordering in Network Processors. In *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, Long Beach, CA, USA, 1–10. https://ieeexplore.ieee.org/abstract/document/4228015

[22] J. Gross, I. Ganga, and T. Sridhar. 2020. Geneve: Generic Network Virtualization Encapsulation. RFC 8926. https://www.rfc-editor.org/rfc/rfc8926.html

[23] Red Hat. 2025. Automatic NUMA Balancing. https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/sect-virtualization_tuning_optimization_guide-numa-auto_numa_balancing. (Accessed: 2025-07-14).

[24] Dave Katz and David Ward. 2010. Bidirectional Forwarding Detection (BFD). RFC 5880. https://www.rfc-editor.org/info/rfc5880

[25] Patrick Kennedy. 2022. Intel IPU Plans Revealed for 800Gbps IPUs in 2025. https://www.servethehome.com/intel-ipu-plans-revealed-for-800gbps-ipus-in-2025-dpu/. (Accessed: 2025-07-14).

[26] Yifan Li, Jiaqi Gao, Ennan Zhai, Mengqi Liu, Kun Liu, and Hongqiang Harry Liu. 2022. Cetus: Releasing P4 Programmers from the Chore of Trial and Error Compiling. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 371–385. https://www.usenix.org/conference/nsdi22/presentation/li-yifan

[27] Jianyuan Lu, Tian Pan, Shan He, Mao Miao, Guangzhe Zhou, Yining Qi, Biao Lyu, and Shunmin Zhu. 2021. A Two-Stage Heavy Hitter Detection System Based on CPU Spikes at Cloud-Scale Gateways. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, DC, USA, 348–358. https://ieeexplore.ieee.org/document/9546443

[28] Nick McKeown. 2023. Intel's commitment to P4. https://groups.google.com/a/lists.p4.org/g/p4-announce/c/frXi_jjmawE. (Accessed: 2025-07-14).

[29] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/3098822.3098824

[30] Tian Pan, Kun Liu, Xionglie Wei, Yisong Qiao, Jun Hu, Zhiguo Li, Jun Liang, Tiesheng Cheng, Wenqiang Su, Jie Lu, Yuke Hong, Zhengzhong Wang, Zhi Xu, Chongjing Dai, Peiqiao Wang, Xuetao Jia, Jianyuan Lu, Enge Song, Jun Zeng, Biao Lyu, Ennan Zhai, Jiao Zhang, Tao Huang, Dennis Cai, and Shunmin Zhu. 2024. LuoShen: A Hyper-Converged Programmable Gateway for Multi-Tenant Multi-Service Edge Clouds. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 877–892. https://www.usenix.org/conference/nsdi24/presentation/pan

[31] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. 2021. Sailfish: accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 194–206. https://doi.org/10.1145/3452296.3472889

[32] 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)*. USENIX Association, Oakland, CA, 117–130. https://www.usenix.org/

conference/nsdi15/technical-sessions/presentation/pfaff
[33] Paul Quinn, Uri Elzur, and Carlos Pignataro. 2018. Network Service Header (NSH). RFC 8300. https://www.rfc-editor.org/rfc/rfc8300.html
[34] Enge Song, Nianbing Yu, Tian Pan, Qiang Fu, Liang Xu, Xionglie Wei, Yisong Qiao, Jianyuan Lu, Yijian Dong, Mingxu Xie, Jun He, Jinkui Mao, Zhengjie Luo, Chenhao Jia, Jiao Zhang, Tao Huang, Biao Lyu, and Shunmin Zhu. 2022. MIMIC: SmartNIC-aided Flow Backpressure for CPU Overloading Protection in Multi-Tenant Clouds. In *2022 IEEE 30th International Conference on Network Protocols (ICNP)*. IEEE, Lexington, KY, USA, 1–11. https://ieeexplore.ieee.org/document/9940340
[35] Chengkun Wei, Xing Li, Ye Yang, Xiaochong Jiang, Tianyu Xu, Bowen Yang, Taotao Wu, Chao Xu, Yilong Lv, Haifeng Gao, Zhentao Zhang, Zikang Chen, Zeke Wang, Zihui Zhang, Shunmin Zhu, and Wenzhi Chen. 2023. Achelous: Enabling Programmability, Elasticity, and Reliability in Hyperscale Cloud Networks. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 769–782. https://doi.org/10.1145/3603269.3604859
[36] Yifan Yuan, Ren Wang, Narayan Ranganathan, Nikhil Rao, Sanjay Kumar, Philip Lantz, Vivekananthan Sanjeepan, Jorge Cabrera, Atul Kwatra, Rajesh Sankaran, Ipoom Jeong, and Nam Sung Kim. 2024. Intel Accelerators Ecosystem: An SoC-Oriented Perspective : Industry Product. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 848–862. https://doi.ieeecomputersociety.org/10.1109/ISCA59077.2024.00066
[37] Shunmin Zhu, Jianyuan Lu, Biao Lyu, Tian Pan, Chenhao Jia, Xin Cheng, Daxiang Kang, Yilong Lv, Fukun Yang, Xiaobo Xue, Zhiliang Wang, and Jiahai Yang. 2022. Zoonet: a proactive telemetry system for large-scale cloud networks. In *Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies* (Roma, Italy) *(CoNEXT '22)*. Association for Computing Machinery, New York, NY, USA, 321–336. https://doi.org/10.1145/3555050.3569116

Appendices

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

A Basic Pipeline

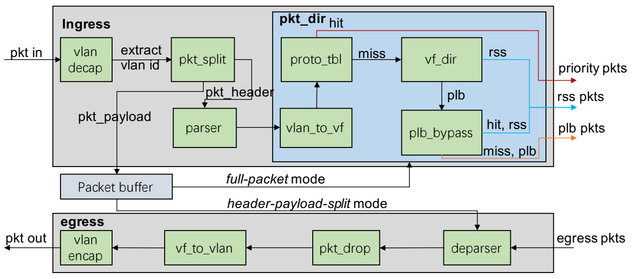

Figure A.1: Basic Pipeline

The basic pipeline (BP) handles packet reception and transmission, parsing, and deparsing, as shown in Fig. A.1. Utilizing SR-IOV, multiple virtual functions (VFs) are created on a single physical network interface card (NIC) for different GW pods. Albatross employs VLAN tags to differentiate VFs. Such VLAN tags will be applied by uplink switches when packets are sent to Albatross. Consequently, the BP's ingress and egress involve VLAN encapsulation and decapsulation. Albatross supports both full-packet and header-payload-split modes. In the ingress path, a `pkt_split` module divides packets into `pkt_header` and `pkt_payload`. For full-packet mode, the complete packet is reassembled before being sent to the CPU; in header-payload-split mode, only the `pkt_header` is forwarded to the CPU, with the packet being reassembled at the egress deparser. The header-payload-split mode is advantageous for large packets, especially Jumbo frames, as it significantly reduces PCIe bandwidth pressure between the NIC pipeline and the CPU, enhancing the throughput of GW pods. The packet mode can be configured individually by each GW pod.

B Containers

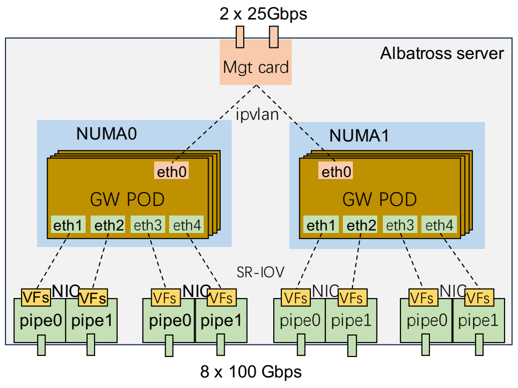

Figure B.1: Albatross Virtualization.

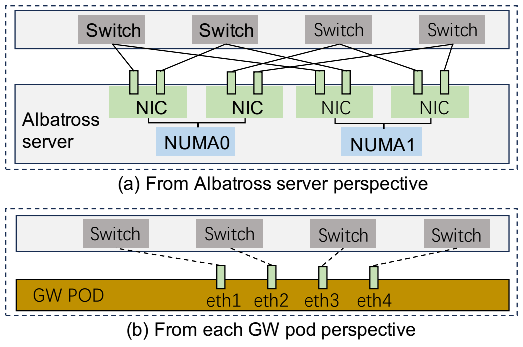
Switch
NIC
Albatross server
NUMA0
NUMA1
(a) From Albatross server perspective
GW POD
eth1
eth2
eth3
eth4
(b) From each GW pod perspective

Figure B.2: Robust GW pod connection design. Switch is the direct-connect switch of Albatross.

High availability design for gateway containers. Albatross hosts multiple types of cloud gateways, each supporting a massive number of tenants and their services. Therefore, the high availability of GW pods is crucial, ensuring that redundant links can continue to provide normal service even if some links are damaged. To achieve this, as shown in Fig. B.1, each container on Albatross supports using four ports from two NICs within the same NUMA node for traffic transmission. Our goal is to make these four network connections fully independent, such that if any NIC or its associated connection fails, the other connections remain unaffected. To achieve this high availability goal, we adopted the interconnection approach shown in Fig B.2(a). This approach ensures that the GW pod's four connections are routed through independent link channels to four different switches, as illustrated in Fig. B.2(b). In this configuration, a failover in any switch affects only one network connection of the GW pod. To implement this architecture, we designed two independent pipelines on each NIC, with each pipeline handling one 100Gbps port.