PacketExpress: Fully Exploiting Large MTUs for Internet Traffic in Private Networks

Junghan Yoon
Seoul National University

Youngmin Choi
KAIST

Juyoung Park
Seoul National University

Daehyeok Kim
The University of Texas at Austin

Changhoon Kim
Unaffiliated (now at Google)

KyoungSoo Park
Seoul National University

Abstract

Network bandwidth continues to scale rapidly, yet Internet data transmission performance remains constrained by the legacy 1500 B MTU. This small MTU translates high bandwidth into high packet rates that strain CPU processing at middleboxes and end hosts. While increasing the MTU could substantially improve performance, coordinating upgrades across arbitrary Internet paths is impractical.

This paper presents PacketExpress, a packet processing architecture that enables networks to leverage large MTUs for Internet traffic without requiring modifications to neighboring networks. MTU-translating gateways at network borders dynamically aggregate incoming small packets into larger packets for efficient processing, then segment them back when forwarding externally. We present PXIO, a packet processing stack that leverages NIC offload capabilities to achieve high throughput. We introduce F-PMTUD, which determines the path MTU within a single round-trip time without relying on ICMP. For UDP, we present PX-caravan, a tunneling mechanism that encapsulates multiple packets to benefit from large MTUs while preserving packet boundaries. Our prototype achieves 1.47 Tbps throughput with 8 CPU cores while converting over 90% of 1500 B packets into 9000 B packets, improving middlebox performance by up to 5.1× and end-host performance by up to 2.5×.

CCS Concepts

• **Networks** → **Routers**; **Wired access networks**; **Mobile networks**; **Middle boxes / network appliances**; **Network performance analysis**.

Keywords

WAN, MTU, PMTUD, NIC, Cellular network

ACM Reference Format:

Junghan Yoon, Youngmin Choi, Juyoung Park, Daehyeok Kim, Changhoon Kim, and KyoungSoo Park. 2026. PacketExpress: Fully Exploiting Large MTUs for Internet Traffic in Private Networks. In *ACM SIGCOMM 2026 Conference (SIGCOMM '26), August 17–21, 2026, Denver, CO, USA*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3789240.3829113

SIGCOMM '26, Denver, CO, USA

ACM ISBN 979-8-4007-2467-1/26/08
https://doi.org/10.1145/3789240.3829113

1 Introduction

The maximum transmission unit (MTU) defines the largest packet size that can be transmitted intact over a layer-3 network. Internet applications typically restrict packet sizes to the *path MTU*, the minimum MTU along the entire route from source to destination, because packets exceeding a network's MTU undergo IP fragmentation and reassembly, which degrades transmission performance. In wide-area networks, the common path MTU remains 1500 bytes (B) [46] (see §2.1), corresponding to the maximum payload size of a frame specified by the original 10 Mbps Ethernet [90]. Given Ethernet's sustained prevalence as a layer-2 protocol, this four-decade-old standard is likely to persist for the foreseeable future.

However, the persistence of this small MTU creates a growing challenge for modern networks. Emerging applications such as 8K/16K video streaming [56, 101], cloud-based virtual reality (VR) [78, 102], and holoportation [82] now demand transmission rates ranging from hundreds of Mbps to over 1 Gbps per flow. While modern NICs [10, 18, 79] can deliver bandwidth in the hundreds of Gbps and off-the-shelf switches [23, 30, 43] can handle traffic in the tens of Tbps, achieving these high bandwidths with a fixed 1500 B MTU requires a proportional increase in packet rate. For instance, sustaining 100 Gbps with 1500 B packets requires processing over 8 million packets per second. Unfortunately, CPU performance improvements have largely stagnated [13, 61], making it increasingly difficult for CPUs to keep pace with these packet rates. This creates a processing bottleneck for middleboxes and end hosts that must perform per-packet operations, particularly for middleboxes such as carrier-grade NATs, firewalls, L4 load balancers, and 5G UPFs that require multiple rule table lookups per packet.

Adopting a large MTU can substantially reduce CPU demand for bandwidth-intensive applications. A large MTU increases the payload-to-header ratio and improves the efficiency of the packet-processing stack by reducing DMA overhead and lowering packet counts. This advantage is particularly pronounced for systems that perform multiple operations per packet, such as rule table lookups in middleboxes.

However, adopting a large MTU across arbitrary Internet paths is challenging, as upgrading the MTU of every network simultaneously is virtually impossible. Instead, we take a pragmatic approach that enables selective upgrades for networks willing to deploy a large MTU. In fact, cellular access and core networks, data centers, and large enterprise networks—typically operated by a single administrative domain—can adopt a large MTU and benefit from large packets within their networks. However, the current practice of configuring a large MTU for all links in a private network is

insufficient because the inter-network traffic entering or leaving the private network must still be forwarded in packet sizes smaller than 1500 B, even within the private network, curbing the benefits of large packets. Our work addresses this exact problem and allows the private network to benefit from large packets for the entirety of the traffic within the network.

To achieve this goal, we introduce PacketExpress (PX), an in-network packet processing architecture that dynamically adjusts packet sizes as they enter and exit a "beneficiary network" (b-network) configured for a large MTU. Our key idea is simple: PX introduces flow-aware, packet-level middleboxes, called *PX Gateways* (PXGW), at the perimeter of the b-network to transparently adjust packet sizes between the b-network and neighboring networks. For example, if network X uses a 9 KB MTU while neighboring network Y retains the legacy 1500 B MTU, PXGW merges incoming 1500 B packets from network Y into 9 KB packets for network X. Conversely, PXGW segments outgoing 9 KB packets from network X into 1500 B packets before forwarding them to network Y. Within network X, all devices (switches, routers, middleboxes, and end hosts) are configured to operate with the large MTU, benefiting from increased packet size and reduced packet counts. This approach is transparent to neighboring networks, requiring no MTU upgrade outside the b-network.

Realizing PX entails several technical challenges. First, PXGW must achieve Tbps-scale throughput and low latency with high packet-merging effectiveness, yet copying packet payloads during merging saturates memory bandwidth, and interleaved packets from multiple flows degrade hardware offload effectiveness. Second, determining the path MTU (PMTU) to the destination is necessary to correctly segment large outgoing packets, but existing ICMP-based Path MTU Discovery (PMTUD) [29] is often unreliable due to ICMP black holes [12, 48] and Packetization Layer PMTUD (PLPMTUD) requires multiple RTTs for convergence. Third, dynamically resizing UDP packets poses a challenge because UDP datagrams are atomic messages with defined boundaries that applications rely on and cannot be transparently split or merged without breaking application-level semantics.

To address these challenges, we present three key techniques. First, we introduce PXIO, a high-performance packet processing stack that leverages off-the-shelf NIC offload capabilities such as large receive offload (LRO), TCP segmentation offload (TSO), scatter-gather DMA, and hardware flow steering to achieve high throughput while maintaining low latency (§4). Second, we devise F-PMTUD, a fragment-based PMTU discovery mechanism that determines the PMTU within a single RTT without relying on ICMP, enabling fast and reliable path MTU discovery (§5.2). Third, we introduce PX-caravan, a tunneling mechanism that encapsulates multiple UDP packets from the same flow or to the same destination into a single enlarged packet, allowing UDP traffic to benefit from large MTUs without violating end-to-end semantics (§6).

We implement PX as a set of packet processing APIs using DPDK [33] and prototype PXGW as a lightweight application using these APIs [99]. Our evaluation demonstrates that PXGW achieves up to 1.47 Tbps of packet forwarding throughput with only 8 CPU cores while dynamically converting over 90% of inbound 1500 B TCP packets into 9 KB packets with an average delay of 20 μs. We find that large packets enabled by PXGW improve the performance of unmodified middleboxes by 4.3× to 5.1×, while enhancing end-host performance by 1.3× to 2.5×. Furthermore, PX can be independently applied to middleboxes even without deploying PXGW, yielding speedups ranging from 2.0× to 7.7× through dynamic packet resizing within the middlebox.

2 Background & Motivation

2.1 Prevalence of 1500-byte MTU

We examine the prevalence of MTU values on the Internet by measuring the PMTU to popular websites using the Cloudflare Radar Top 1 Million Domains list [24]. Of the 570,045 reachable domains, 93.7% exhibit a PMTU of 1480 B or larger, specifically, 89.0% at 1500 B and 4.3% at 1480 B. Conversely, only 0.25% of paths show an MTU smaller than 1000 B. These findings underscore the continued dominance of the 1500 B MTU in the modern Internet.

The 1500 B MTU originates from the maximum payload size defined in early Ethernet. The seminal paper on Ethernet in 1976 [72] proposed 500 B, but the first Ethernet standard [90] in 1980 specified 46 B to 1500 B, which endures in the latest IEEE 802.3 standard [59]. While the minimum of 46 B was necessary for reliable collision detection [60], what was the rationale behind the 1500 B maximum?

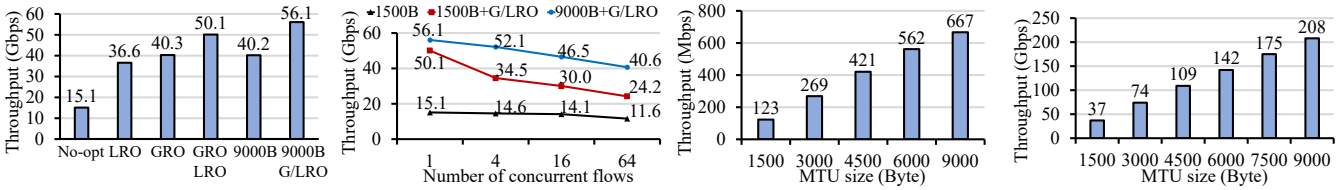
Bob Metcalfe, an inventor of Ethernet, recalls that 1500 B was chosen to align with the disk sector size plus overhead for Xerox-D machines [31], allowing the OS to process data at sector boundaries.[1] He explained the general strategy: increasing the payload size would lead to higher forwarding delays and greater susceptibility to packet losses due to channel errors, while a smaller size would reduce efficiency. Interestingly, the 1500 B limit had no intrinsic connection to Ethernet's fundamental operation, yet it remains the de-facto MTU across the entire Internet today.

We argue that this historical constraint is no longer relevant in modern networks. Today's *switched* Ethernet allows concurrent point-to-point communications, eliminating contention-driven delays of the original shared-medium Ethernet. While larger frames may increase head-of-line blocking, network bandwidth has increased by several orders of magnitude: in a 400 Gbps datacenter network, transmitting a 9 KB packet takes only 0.18 μs, and even a 64 KB packet takes just 1.31 μs. Moreover, bit error rates (BER) have substantially improved in modern NICs and switches. The IEEE standard mandates the 64 B frame loss rate to be less than 6.2×10^{-11} for 200/400 GbE-SR/VR [58]—one frame loss per 1261 GB even with a 64 KB MTU. While larger MTUs increase retransmission overhead per error, the impact remains minimal given the low probability. Given these technical advances, the question shifts from whether large MTUs are feasible to where they can be most effectively deployed.

2.2 Beneficiary Networks of Large MTU

While any network can in principle deploy and benefit from large MTUs, cellular and datacenter (DC) networks are particularly well-positioned to do so. Both environments employ software-based middleboxes, such as 5G UPFs, carrier-grade NATs, firewalls, and load

[1] In personal communication with Bob Metcalfe on April 22, 2024.



(a) Impact of G/LRO (single flow)

(b) Impact of concurrent flows

(c) Impact of MTU size for WAN connection (single flow)

(d) Impact of MTU size on the 5G UPF performance

Figure 1: Effectiveness of large packets

balancers, at their network borders. Since these middleboxes primarily process packet headers rather than payloads, their processing load can be significantly reduced by handling fewer packets for the same amount of data. Even for systems that reassemble flow content or scan payloads, larger MTUs improve performance by reducing interrupt handling and DMA overhead.

Importantly, no fundamental hardware limitation prevents large MTU deployment in these networks. Cellular access networks, which constitute a major form of the last mile, support large MTUs: the 3GPP standard [5] specifies a maximum Packet Data Convergence Protocol (PDCP) payload size of 9 KB. In DC networks, commodity switches, routers, and NICs, even at 1 GbE, commonly support MTUs up to 9 KB, and jumbo frames are already widely employed within the network.

These networks also host emerging bandwidth-intensive applications that would particularly benefit from larger MTUs. According to 5G service requirements [3], high-quality virtual reality (VR) and augmented reality (AR) content requires up to 1 Gbps for cloud or edge rendering and up to 10 Gbps for VR headsets that lack processing power to decode compressed data in real-time. Other emerging applications, such as immersive XR, high-fidelity holograms, and digital twins, are projected to require 10 to 50× the peak data rate of 5G networks [7, 92]. Given these demands, we next quantify the concrete performance benefits that large MTUs provide.

2.3 Performance Benefits of Large Packets

To quantify the benefits, we evaluate the effectiveness of large MTUs for traffic endpoints and middleboxes. A key question is whether large packets provide meaningful improvements over existing NIC offloads such as generic receive offload (GRO) and large receive offload (LRO), which systems already employ to reduce per-packet processing overhead.

Benefits to endpoints. Figure 1a demonstrates that enabling GRO and LRO achieves throughput gains comparable to increasing the MTU. With G/LRO turned on, single-flow throughput reaches 50.1 Gbps with a 1500 B MTU—even outperforming the 9 KB MTU without these offloads. This raises a critical question: Is a large MTU truly necessary for traffic endpoints?

We argue that large packets remain beneficial for endpoints for several reasons. First, G/LRO effectiveness degrades rapidly as the number of concurrent flows increases. Figure 1b shows that aggregate throughput drops by 31% with only 4 concurrent flows because interleaved packets from multiple flows reduce aggregation opportunities. In contrast, throughput with larger MTUs degrades by only 7% under the same conditions. Second, while G/LRO only optimizes the receive side, large MTUs benefit both endpoints. Furthermore, a larger MTU (and thus a larger maximum segment size (MSS)) allows the congestion window to scale more aggressively, as the window grows by one MSS per ACK during slow start or per RTT [8, 9] during congestion avoidance. TCP throughput in steady state is also proportional to MSS [70, 83]. For instance, with a 10 ms end-to-end delay and a 0.01% loss rate, a 9 KB MTU outperforms a 1500 B MTU with G/LRO by 5.4× as shown in Figure 1c. Finally, LRO is not yet widely adopted by commodity on-board NICs [17, 42]; while GRO can serve as a substitute, it incurs additional CPU overhead.

Benefits to middleboxes. Large packets provide even greater benefits to middleboxes. We test an open-source cellular UPF from the Open Mobile Evolved Core (OMEC) project [36], which leverages BESS [40], a modular software switch running on DPDK [33]. We generate 800 concurrent TCP flows using iPerf and configure the UPF to use a single CPU core (rule setup details in §7). Figure 1d shows that UPF throughput scales almost linearly with MTU size, achieving 208 Gbps with a 9 KB MTU on a single core—a 5.6× speedup over a 1500 B MTU. Larger MTUs are particularly advantageous since UPFs process only packet headers.

3 Overview of PacketExpress

PacketExpress (PX) allows private networks that have already adopted a large MTU to sufficiently take advantage of that large MTU for Internet traffic, even when external networks continue to use the legacy 1500 B MTU. Figure 2 shows the overall architecture. The key idea is to deploy flow-aware PX Gateways (PXGW) at the network border to translate packet sizes between internal and external MTUs. We refer to the network that employs PX as a **"beneficiary network" (b-network)**. We define iMTU[2] as the internal MTU within the b-network, and eMTU as the external MTUs used by other networks on the Internet.

PXGW performs bidirectional translation: for inbound traffic, it merges contiguous packets in the same flow into a single packet bounded by the iMTU; for outbound traffic, it splits each large packet into multiple eMTU-sized packets. This approach benefits all devices within the b-network: end hosts process fewer packets, middleboxes perform fewer lookups, and switches forward less traffic. Notably, this requires no changes to external networks.

[2]iMTU can reach the maximum IP packet size (64 KB), but we currently use 9 KB because most network devices support this MTU size.

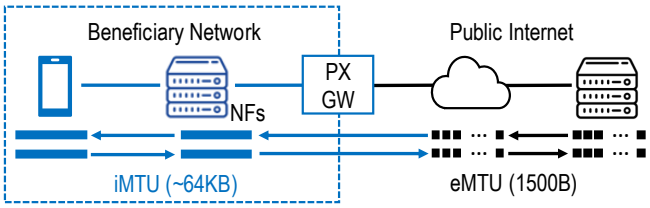


Figure 2: Overall architecture of PacketExpress

3.1 Why Not Layer-3 Translation?

MTU translation can, in principle, be implemented at either the network layer (L3) or the transport layer (L4). L3 translation using IP fragmentation and reassembly [1] appears straightforward, as it directly leverages standard IP functionality. However, this approach introduces fundamental problems that make it unsuitable for PX.

Protocol-level issues. IP fragmentation and reassembly suffer from well-known reliability and security vulnerabilities [15, 52, 68, 89]. IP reassembly operates without retransmission; any fragment loss forces the receiver to discard the entire datagram, causing loss amplification. Fragment buffering enables denial-of-service attacks by exhausting memory with incomplete fragments. The 16-bit IP identification field is vulnerable to collisions at high packet rates, leading to datagram corruption or spurious drops.

Deployment limitations. Beyond these protocol issues, L3 translation has deployment constraints that limit its applicability. IP reassembly at end hosts provides no receiver-side benefit and instead adds processing overhead. In-network IP reassembly can only operate on packets that were explicitly fragmented at L3, not on arbitrary packets. This requires both sender and receiver networks to coordinate identical MTU deployments, restricting benefits to flows between two upgraded networks. IP fragmentation was designed for datagram reachability across heterogeneous links, not for exploiting large MTUs in a partially upgraded Internet.

3.2 Transport-Layer Translation

We therefore turn to transport-layer (L4) translation, which can merge arbitrary packets (not just IP fragments) while preserving end-to-end semantics. The key advantage of L4 translation is that it can operate on normal packets sent at standard MTU sizes, enabling single-side deployment without requiring coordination between networks. For TCP, this is straightforward: byte-stream semantics allow segment boundaries to be modified without violating correctness, enabling PXGW to perform stateless segmentation and reassembly without terminating connections. TCP's end-to-end reliability eliminates loss amplification, sequence numbers avoid ID collisions, and complete headers ensure middlebox compatibility. As TCP reacts to bytes acknowledged rather than packet counts [8, 14], congestion control is naturally preserved.

3.3 Challenges

Despite these advantages, realizing efficient transport-layer translation in practice presents three challenges:

C1: High throughput with high merging effectiveness. PXGW must merge and split packets at multi-Tbps rates while performing flow identification, neighbor matching, and header rewriting. However, a naïve software implementation achieves only 166 Gbps on our 8-core platform, far below the multi-Tbps rates required for modern networks.

C2: PMTU discovery across heterogeneous MTUs. As PX gains wider adoption, certain network paths may support a path MTU (PMTU) exceeding 1500 B. Then PXGW must determine the safe PMTU to avoid IP fragmentation. Unfortunately, existing PMTUD mechanisms [29, 69] are either unreliable due to ICMP black holes or too slow, requiring multiple RTTs (§5.1).

C3: Supporting UDP atomicity constraints. Unlike byte-stream protocols (e.g., TCP), UDP requires each datagram to be delivered atomically, preventing arbitrary packet merging or splitting. Enabling large MTU benefits for UDP while preserving packet boundaries requires a different approach.

3.4 Key Ideas

To address these challenges, PX introduces three key ideas:

I1: High-performance packet processing stack (§4). PXIO is a scalable packet processing stack that leverages NIC hardware offloads (TSO, LRO) and introduces optimizations, including zero-copy merging, dynamic RXQ scaling, and hairpin queues for small flows. PXIO supports both TCP and UDP traffic, vastly outperforming the baseline implementation.

I2: Fast PMTU discovery without ICMP (§5.2). F-PMTUD is a new PMTU discovery protocol that exploits controlled IP fragmentation for probing. It enables PXGW to reliably infer the PMTU in a single RTT without relying on ICMP, avoiding black hole issues while accelerating discovery. F-PMTUD works for both TCP and UDP traffic.

I3: Packet aggregation for UDP (§6). PX-caravan is a tunneling-based mechanism that encapsulates multiple UDP packets into a single large packet for transmission within the b-network, then decapsulates them at the destination to restore their original boundaries. This approach enables PX for UDP traffic with minor modifications at endpoints in the b-network.

3.5 Deployment Model

Deploying PX requires configuring all in-network devices (switches, routers, and middleboxes) within the b-network to use iMTU. This requirement is local to a single b-network and does not require changes in neighboring networks. Cellular and datacenter networks are well-suited as b-networks: central management under one administrative domain lets operators coordinate MTU configuration across the network.

End-host requirements. For TCP traffic, PX can be deployed immediately once PXGW is introduced and iMTU is configured, without requiring modifications to end hosts. For UDP traffic, MTU translation requires minor modifications to the UDP stack at endpoints within the b-network to support PX-caravan encapsulation and decapsulation (§6).

Incremental deployment. During incremental deployment, some endpoints within the b-network may not yet be configured for iMTU. In these cases, PXGW infers the endpoint MTU by inspecting the

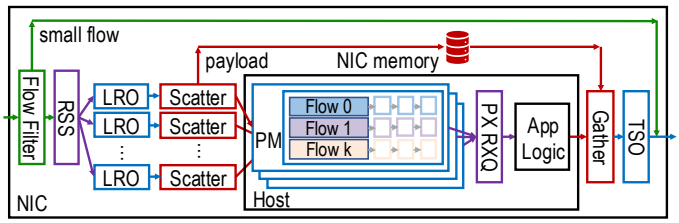


Figure 3: Overall architecture of PXIO

TCP MSS value advertised during the handshake. For flows that advertise legacy MSS values—or where the MSS value is unavailable (e.g., due to asymmetric routing or route changes)—PXGW does not perform MTU translation and forwards packets at their original size. If the receiver MTU later becomes available through explicit signaling or any other control-plane mechanism, PXGW can re-enable MTU translation for subsequent traffic.

4 Scalable Packet Merging Stack

PXGW operates by splitting and merging packets in a flow-aware way. Splitting large packets is straightforward and efficient, so we concentrate on merging, whose goal is to maximize iMTU-bounded ($iMTU_b$) packets while satisfying latency constraints. The challenge is performance: a naïve implementation that copies payloads into contiguous buffers achieves just 166 Gbps on our 8-core platform—8.8× below our optimized PXGW.

To address this challenge, PX introduces **PXIO**, a scalable packet processing stack that leverages NIC hardware capabilities. Figure 3 presents the overall PXIO architecture. PXIO builds upon standard NIC offload features, such as LRO, TSO, and checksum offloading, and adds software packet merging (PM) to aggregate packets that the hardware cannot merge. While hardware LRO coalesces consecutive in-order TCP packets within the same flow, PM handles packets that arrive out of order or are interleaved with other flows.

Achieving high-performance PM requires two components: an efficient core algorithm and targeted optimizations. The core algorithm (§4.1) addresses how to identify flows, find mergeable packets, and balance latency versus merge opportunities. The optimizations tackle three key bottlenecks. First, copying packet payloads into contiguous buffers during merging saturates memory bandwidth, limiting throughput. We address this through zero-copy packet chaining and header-only DMA to minimize data movement (§4.2). Second, when many flows are interleaved in a single RX queue, LRO effectiveness degrades significantly. We address this by dynamically scaling the number of RX queues to reduce per-queue flow contention (§4.3). Third, processing numerous small flows wastes CPU cycles and interferes with merging large flows. We address this by selectively bypassing small flows using NIC hairpin queues (§4.4). Together, these techniques enable PXIO to achieve 1.47 Tbps with 8 cores while converting 93% of inbound packets to 9 KB.

Beyond PXGW, PXIO serves as a general packet processing stack for network middleboxes. Middleboxes using PXIO can dynamically increase inbound packet sizes, reducing the packet count that application logic must process (indicated as App Logic in Figure 3). After processing, packets are resized to conform to the next hop's MTU before forwarding. Since the overhead of adaptive resizing is minimal relative to core application logic, overall performance improves considerably. PXIO preserves standard packet formats on the wire rather than relying on non-standard headers or any brittle protocol behavior, which allows existing middlebox logic to process packets unchanged. We demonstrate these improvements in §7.

4.1 Basic Packet Merging in PXIO

The PM algorithm must efficiently identify which packets belong to the same flow and can be merged while respecting TCP sequence numbers and minimizing latency. This is more challenging than hardware LRO, which operates on consecutive, in-order packets within a single queue. PM must handle packets that arrive out of order or are interleaved with packets from many other flows.

Fast flow identification. PM leverages hardware-generated RSS hashes [34] to achieve O(1) flow lookups. Modern NICs compute a Toeplitz hash over the packet's four-tuple (src/dest IPs and ports) and include it directly in the packet descriptor. By using this hash as the flow table key, PM bypasses the costly O(N) linked-list traversals common in prior approaches [32, 37, 64]. This optimization improves throughput by 10% to 35% depending on the number of concurrent flows.

Bidirectional neighbor identification. Prior packet coalescing approaches [32, 64] consider only one-sided merging. PM improves upon this by identifying merging candidates in both directions. For example, if packet B arrives while packets C and A are already stored, existing schemes might only merge B with its immediate successor or predecessor. In contrast, PM successfully merges all three by recognizing that A fits the gap between C and B. This bidirectional search significantly increases merge opportunities for reordered packets.

Adaptive delayed merging. To maximize merging efficiency, PM may briefly delay forwarding an incomplete packet to await subsequent segments. However, fixed timeouts [38, 64] are suboptimal: short timeouts limit coalescing opportunities, while long timeouts increase latency. Instead, PM adopts dynamic per-flow timeouts by tracking per-flow throughput and packet inter-arrival time. This adaptive strategy merges up to 90% of incoming packets with an average processing delay of 17 μs (see Figure 11a in §7.1). Note that this delay is non-cumulative, since the delayed merging process for each packet is fully pipelined. It is incurred only once and does not grow with flow size or packet count. Because PX targets Internet-scale, bandwidth-intensive applications that benefit from larger MTUs and typically have millisecond-scale latency budgets, this delay is negligible.[3] The detailed algorithm and its statistical model are described in Appendix A. After merging, PM applies TSO to segment merged packets into $iMTU_b$ packets for transmission.

4.2 Memory Bandwidth Optimization

To avoid memory bandwidth bottlenecks, PXIO combines two complementary techniques: zero-copy packet chaining, which avoids redundant payload copies, and header-only DMA, which transfers only packet headers to the host memory.

[3] Operators can also let latency-sensitive, low-bandwidth flows bypass PX when needed.

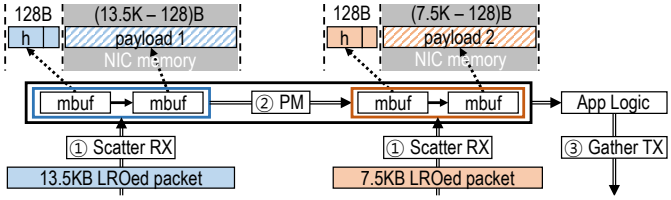


Figure 4: Logical sequence of header-only packet merging

Packet chaining with gather TX. A straightforward implementation of the PM algorithm would copy packet payloads into contiguous buffers to create merged packets. However, this results in poor throughput as most CPU cycles are stalled on memory access. PXIO exploits scatter-gather DMA to avoid this overhead. In particular, gather TX enables logically chained packets to be transmitted without linearization. PXIO internally maintains only a logical chain of packet headers without payload copying, which is especially beneficial for packet-level middleboxes that access only packet headers.

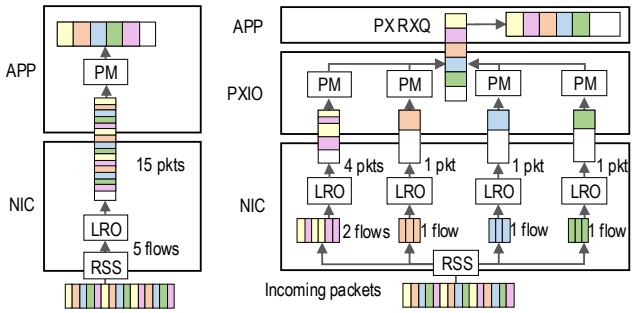
Header-only DMA with scatter RX. While zero-copy packet chaining avoids copying payloads in host memory, it still requires transferring entire packets from NIC to host memory via DMA. At high packet rates, this DMA transfer itself consumes significant memory bandwidth. To further reduce memory traffic, PXIO adopts header-only DMA [86] that transfers only the first 128 bytes of each packet into host memory while keeping the remaining payload in NIC memory. Specifically, PXIO allocates NIC memory at initialization and registers it as an external pool for the scatter RX queue. As illustrated in Figure 4, ① scatter RX splits each received packet across two `mbufs`: a 128 B host buffer containing the header and a NIC memory buffer containing the remaining payload. ② These chained `mbufs` can be extended during packet merging and directly passed to the application. After application processing, ③ gather TX combines the header with logically merged payloads into a single iMTU_b packet for transmission.

Header-only DMA substantially improves throughput, but we find two drawbacks. First, it incurs extra CPU overhead, such as additional pointer initialization and management. As a result, when CPU rather than memory bandwidth is the bottleneck, header-only DMA degrades performance. Second, it requires sufficient on-NIC packet-store capacity to hold in-flight packet payloads. In our prototype, a firmware limit caps the NIC's on-device memory for this purpose at 256 KB, so receiving packets whose total in-flight size exceeds this capacity can cause overflow. The required capacity is roughly link bandwidth times processing latency: for a 400 Gbps NIC, this amounts to about 4–5 MB under our conservative latency estimate.[4] While the overflow condition reflects an implementation constraint rather than a fundamental limitation of header-only DMA, we keep this feature experimental.

4.3 Scaling with Multiple RX Queues

The previous optimizations relieve the memory bandwidth bottleneck, but flow interleaving creates a new one. When many flows are mixed into a single RX queue (RXQ), LRO can no longer merge packets effectively.

[4]Prior work [86] reports that ConnectX-6 has 4 MB of on-device memory, which is close to this requirement, and NVIDIA architects have confirmed that further increase is feasible.



(a) PM on single RXQ **(b) PXIO using 4 RXQs**

Figure 5: Comparison between single RXQ and multi-RXQs

Modern NICs employ multiple RXQs where LRO is applied to each queue after packets are steered by RSS. Increasing the number of RXQs improves LRO efficiency by reducing the flow count per RXQ. Figure 5 illustrates this: with a single RXQ (left), 15 packets from 5 flows arrive interleaved, preventing LRO from merging any packets. With 4 RXQs (right), RSS distributes flows across queues, enabling LRO to merge packets within each queue before software PM processes them. Moreover, because packets are already partially grouped by flow within each RXQ, packet merging (PM) in software is also accelerated, as it operates on fewer interleaved packet streams. However, having too many RXQs increases polling overhead, so the optimal count depends on active flows and traffic volume. This differs from recent RX ring management work like shRing [87] and rxBisect [88], which address leaky DMA rather than optimizing RXQ count for LRO efficiency.

Dynamic RXQ count adjustment. Changing the number of RXQs requires a full NIC restart, incurring significant overhead. Instead, PXIO allocates the maximum number of RXQs at initialization and dynamically updates RSS steering rules to redirect traffic to a subset, effectively adjusting active RXQs. There are two ways to modify a steering rule: *evict-and-insert* or *insert-and-evict*. The former allows arbitrary RXQ counts but during the ~42 ms transition period, packets spread over all RXQs, causing severe fluctuations. PXIO adopts insert-and-evict, which exploits distinct priority levels for co-existing rules, ensuring atomic transitions.

PXIO employs five effective RXQ configurations: the default full-RXQ configuration and four rule-selected subsets with 2/8/32/128 RXQs. Among the rule-selected subsets, smaller subsets have higher priority, so inserting a new rule reduces the count, while evicting increases it. Each RXQ configuration is partitioned per core so that each worker thread polls a dedicated RXQ range. Figure 6 shows an example of how subsets are partitioned across threads. When the RXQ count is reduced from 16 to 8, only the first four RXQs per thread (0 to 3 and 8 to 11) become active. During scaling down, PXIO continues polling the old subset until all packets are drained, with rule updates offloaded to a dedicated manager thread via an asynchronous API (§4.4).

Finding the proper RXQ subset. Ideally, each active flow would occupy a dedicated RXQ to minimize flow interleaving. However, dedicating an RXQ to every flow is infeasible as the global flow count can far exceed available RXQs, though most flows are inactive within small time windows. Instead, PXIO tracks the *active*

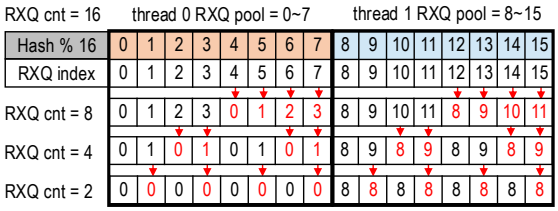


Figure 6: Partitioning of RXQs (2/4/8/16 RXQs) to two threads

flow count (flows observed within a single RX batch) and triggers RXQ upscaling when this count exceeds the RXQ count. Conversely, downscaling is triggered when the active flow count falls below 10% of the RXQ count, a conservative threshold to prevent oscillation. This metric is averaged over a configurable time frame (e.g., 1 second).

Unified API to access multiple RXQs. Directly managing dynamic RXQ counts introduces significant development complexity. To address this, PXIO provides a unified RXQ API (`px_eth_rx_burst()`) that abstracts multi-RXQ operations and eliminates management overhead. Figure 5b shows an example where four underlying RXQs are encapsulated within a single unified PX RXQ. PXIO transparently fetches packets from these queues, merges them, and passes the resulting pointer batch to the application.

4.4 Handling Small Flows

While multiple RXQs mitigate interleaving, the highly skewed nature of Internet traffic presents a distinct challenge: a vast majority of "mice flows" consume CPU resources without benefiting from packet merging, thereby reducing overall system efficiency. Given that most flows are small while a fraction of "elephant flows" accounts for the bulk of traffic [39], this imbalance hinders both LRO and packet merging performance. To address this, PXIO leverages the hairpin queue feature of modern NICs [81], where specialized RXQs directly forward packets to paired TXQs within the hardware without DMA. Recent ConnectX NICs [76, 77, 79] support up to 64 K hairpin rules with flexible match-action entries. By offloading these small flows to hairpin queues, PXIO conserves CPU cycles for large flows and maintains robustness even under workloads with a massive number of concurrent connections (see Figure 11c).

Opt-in policy. Since small flows typically outnumber large flows, the 64K hairpin rule limit may be insufficient to offload all of them. PXIO adopts an opt-in policy: by default, all packets are directed to hairpin queues for on-NIC processing, while only those belonging to target services or flows (e.g., bandwidth-intensive VR or video streaming) are DMA'ed to the packet merging cores for additional processing. At runtime, applications can dynamically select such target services or flows and onload them via the PXIO API.[5]

Asynchronous API. Unfortunately, dynamic rule updates incur high overhead, mainly due to NIC firmware communication delays. On ConnectX NICs, inserting or evicting a per-flow steering rule via the DPDK `rte_flow` API takes over 330 μs [51, 55] or 3K updates/s even with multiple NICs. Because the DPDK API is synchronous,[6]

[5]Automated heavy-hitter detection [39, 97, 98] and a remote control channel such as PFCP [4] can also be applied, but are beyond the scope of this work.

[6]The latest version provides an asynchronous API, but it is limited to certain rule types and depends on vendor-specific implementations.

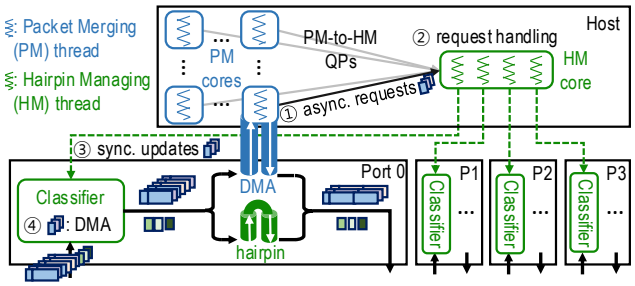


Figure 7: Leveraging hairpin queues to bypass host processing

rule updates block packet processing of PM threads, which degrades the forwarding throughput.

To overcome this limitation, PXIO provides a vendor-agnostic API, `px_flow()`, which enables asynchronous rule updates while hiding communication delays from worker threads. As shown in Figure 7, PXGW dedicates a CPU core to hairpin management (HM) across all NICs. ① Each PM thread asynchronously submits match-action entries via a lockless queue pair (QP) – one for submission and the other for completion notification – to an HM thread on the dedicated core. ② The HM thread accepts these request entries and ③ performs synchronous updates to the NIC ports. While an HM thread blocks during NIC communication, other HM threads continue execution via context switching. This design achieves 12K rule updates per second with four NICs without degrading packet forwarding throughput.

5 Robust Path MTU Discovery

PXGW segments outbound packets to fit the PMTU of the route to avoid IP fragmentation. It infers PMTU on a per-flow basis using the TCP MSS advertised by end hosts outside the b-network. If MSS is unavailable or exceeds the next hop's eMTU, PXGW falls back to eMTU or default values. This mirrors standard end-host behavior and remains effective in today's Internet, where MTUs are typically configured conservatively (e.g., below 1500 B).

However, this heuristic may prove insufficient as PX adoption grows. First, if PXGW is deployed at both ends of a path, both end nodes may advertise large MSS values based on their local iMTUs, potentially exceeding the true PMTU. One fix is to clamp outgoing MSS to the de-facto standard MTU [25], but this would limit large-MTU coverage and reinforce Internet MTU ossification. Second, as PX grows more popular, the actual PMTU may exceed legacy MTU limits, making conservative defaults obsolete.

5.1 Limitations of Existing Path MTU Discovery

A more principled solution is path MTU discovery (PMTUD) [28, 29] or packetization layer PMTUD (PLPMTUD) [35, 69]. However, neither mechanism is sufficient for fast and robust path MTU discovery at PXGWs.

PMTUD. Classical path MTU discovery (PMTUD) [28, 29] relies on the network never fragmenting probe packets—via the "Don't Fragment" (DF) bit in IPv4, and inherently in IPv6, where routers never fragment. If a router along the path has a smaller MTU, it drops the packet and returns an ICMP message—"Fragmentation Needed" in IPv4 or "Packet Too Big" in IPv6, hereafter PTB. On

receiving the PTB packet, the sender reduces its packet size to the next-hop MTU it reports and continues probing. In practice, however, PMTUD works poorly for two reasons. First, PTB packets often omit a valid next-hop MTU or are lost to ICMP black holes [12, 48], where routers or firewalls silently drop them. Second, under equal-cost multi-path (ECMP) routing, the PTB message may never reach the node that sent the oversized packet. Because an intermediate router generates the PTB with its own address as the source IP—not the destination's—the message's 5-tuple differs from the original flow's, so ECMP may steer it to a different node [67].

PLPMTUD. To address these drawbacks, Packetization-Layer PMTUD (PLPMTUD) [35, 69] relies on reliable transport (e.g., TCP or SCTP) instead of PTB messages. The sender transmits progressively larger packets, and the receiver acknowledges them; an ACK signals that the corresponding packet size is safe to use. The algorithm is not always straightforward, however. Suppose a sender transmits a packet larger than the path MTU followed by smaller ones: the oversized packet is dropped, but the smaller ones trigger duplicate ACKs in TCP, and the sender must infer the path MTU from this indirect signal. Moreover, because PLPMTUD discovers the PMTU by trial-and-error probing, raising the estimate to a larger value may take multiple RTTs. Thus, while PLPMTUD sidesteps ICMP-related problems, its deployment remains limited—both by the algorithm's complexity and by the repeated probing its trial-and-error process requires [71]. There is currently no explicit guide for enabling TCP PLPMTUD on operating systems other than Linux [94]. These challenges are amplified for in-network devices such as PXGW, which cannot easily track endpoint transport state or observe end-to-end feedback.

5.2 Fragment-based PMTUD

We propose **fragment-based PMTUD (F-PMTUD)**, a protocol for fast and accurate PMTU discovery between PXGWs. The key insight is to actively exploit IP fragmentation for probing rather than for data delivery. While IP fragmentation introduces reliability and security issues in data delivery [15, 52, 68, 89], using it only for explicit probing avoids these pitfalls. F-PMTUD operates between a source PXGW (S-PXGW) and a destination PXGW (D-PXGW).[7] S-PXGW sends a probe UDP packet to the end node using a well-known port with a probe size equal to the eMTU of the next hop. If the actual PMTU is smaller than eMTU, routers fragment the probe en route. D-PXGW receives the fragmented packets and reports all fragment sizes back to S-PXGW. S-PXGW infers the PMTU as the largest fragment size that arrives. If no D-PXGW exists, the end node drops the fragments. Figure 8 shows an example. S-PXGW sends two probe packets separated by some interval to mitigate potential packet loss. Upon receiving all fragments, D-PXGW reports their sizes back, and S-PXGW determines the PMTU as the largest fragment (1000 B in this case). Since network routes can change dynamically, PXGW periodically reruns probing (e.g., once every 10 minutes) or when it receives ICMP error messages.

Fast PMTU lookup. Once F-PMTUD discovers the PMTU for a flow, PXGW stores it in a per-flow table. Since PXGW queries this table for every outbound packet, efficient lookup is critical. We exploit the fact that 1500 B PMTU remains dominant. The prober maintains a counting Bloom filter tracking only flows with non-1500 B PMTUs. In the common case, the Bloom filter lookup fails, indicating the flow uses 1500 B PMTU and avoiding a full table lookup. Only upon a Bloom filter hit does the prober query the table for the actual PMTU. This yields a negligible false-positive rate (e.g., 0.0047% with 3 hash functions and 64 K counters). To clean up the stale entries in the Bloom filter, the F-PMTUD prober periodically removes entries that have been inactive for a configured period.

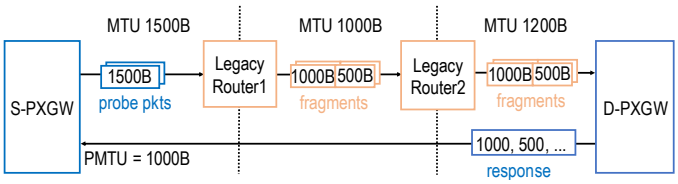


Figure 8: F-PMTUD with legacy routers.

Fallback mechanism. In rare cases where F-PMTUD fails (0.02% of cases, §7.4), PXGW falls back to a default PMTU derived from MSS or eMTU. When both are unavailable, PXGW uses a conservative value that minimizes fragmentation (e.g., 1240 B used by macOS/FreeBSD,[8] and 1280 B as IPv6 minimum MTU [27]).

Correctness and security. F-PMTUD relies on four assumptions: (a) routers fragment packets exceeding their next hop MTU, (b) fragmented packets traverse routers successfully, (c) neither routers nor D-PXGW are compromised, and (d) no counterfeit responses are injected by off-path adversaries. The first two are standard IP behavior, and our tests (§7.4) confirm that fragmented packets pass through. For off-path attackers (d), S-PXGW and D-PXGW can use challenge-response verification. Detecting compromised D-PXGWs or intermediaries (c) requires authentication, which we leave as future work. Even without full authentication, F-PMTUD provides security properties comparable to ICMP-based PMTUD.

Advantages. F-PMTUD offers several advantages over existing PMTUD and PLPMTUD. First, it avoids ICMP black hole issues by not relying on ICMP responses. Second, it typically determines the PMTU within a single RTT, making it significantly faster than existing schemes. Even when routes change due to network updates, it can quickly re-learn the PMTU.

6 Packet Aggregation for UDP

UDP requires protocol data units (PDUs) to be delivered atomically to preserve application-layer semantics. As a result, intermediate nodes cannot arbitrarily split or merge UDP packets in the same manner as TCP. This presents a challenge: how can UDP traffic benefit from large MTUs when packet boundaries must be preserved?

To address this, we design **PX-caravan**, a tunneling-based mechanism that allows UDP traffic to benefit from large MTUs while preserving packet atomicity. The key idea is to encapsulate multiple UDP packets into a single large packet for transmission within the b-network, then decapsulate them at the destination before processing them. This approach requires minor kernel modifications at b-network endpoints to preserve the standard UDP receive interface.

[7] While F-PMTUD deployment is orthogonal to PXGW, we couple them to streamline the deployment process.

[8] `sysctl net.inet.tcp.pmtud_blackhole_mss`

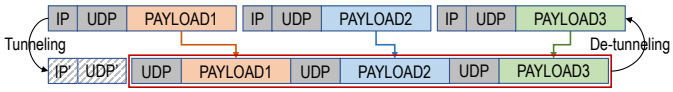


Figure 9: PX-caravan format

Each `recvfrom()` call returns one segment, and each `recvmsg()` call returns multiple segments, from a PX-caravan without additional copies, while the kernel internally tracks the next-segment offset.

Figure 9 shows the format of a PX-caravan packet. Unlike existing tunneling mechanisms (e.g., PPTP/GRE [63, 103], GTP [2]) that encapsulate a single packet, PX-caravan encapsulates multiple UDP packets within a single payload. The structure consists of a common UDP/IP header followed by the concatenated original segments. The common header carries the flow's 5-tuple and the total length of the combined segments, with a mark in the IP QoS field to identify the packet as a PX-caravan frame.

In-network devices (e.g., switches or middleboxes) process the packet using the common header while treating the encapsulated packets as a single payload. This allows the large packet to traverse the b-network efficiently, reducing packet counts and processing overhead. When the PX-caravan packet arrives at the destination, the receiver (end host or PXGW) detects the caravan type from the IP QoS field and decapsulates the individual packets before delivering them to the responsible sockets or forwarding them to the next hop.

Note that UDP packet merging is optional. Certain security middleboxes may require inspection of individual inner segments; if these middleboxes cannot be updated to support PX-caravan, this feature can be disabled.

Application to QUIC. QUIC [45, 57], which runs over UDP, can benefit from PX-caravan in the same manner as other UDP flows. Since QUIC encrypts most transport-layer fields [95], including packet numbers and stream information, PX cannot perform transport-aware packet resizing for QUIC as it does for TCP. However, PX-caravan can encapsulate multiple QUIC packets, allowing the UDP/IP stack to process multiple segments in a larger packet, reducing per-segment overhead.

7 Evaluation

We evaluate PX to answer four questions: (1) Does PXGW achieve high throughput while dynamically converting packet sizes (§7.1)? (2) Does PXGW improve the performance of end nodes and network middleboxes (§7.2)? (3) Can standalone middleboxes benefit from PXIO APIs without deploying PXGW (§7.3)? (4) Is F-PMTUD feasible for deployment in the current Internet (§7.4)?

Implementation. The PXIO APIs comprise 20,125 lines of C code implemented using DPDK 20.11. We implement PXGW as a simple L3 gateway with 1,003 lines of C code atop PXIO. To parse PX-caravan packets at the receiver, we add 303 lines to the UDP stack in Linux kernel 5.15.179. Our PX prototype is publicly available [99].

NFs. We evaluate three NFs typically located at a network border: a 5G UPF, a NAT, and an L4LB. For the UPF, we use Intel's open-source UPF-EPC [36] configured with 4 M PDRs, 500 K QERs, and 500 K FARs.[9] We set per-flow QoS rules with 10 Gbps maximum bit rate (MBR) and 2 Gbps guaranteed bit rate (GBR). For the NAT, we use an SNAT on mOS [47] that monitors per-flow state to prevent attacks such as over-billing [41] or spamming [85]. For the L4LB, we use iQIYI's DPVS [44], a DPDK version of Alibaba's LVS [6], which is widely used as a load balancer in edge networks.

Testbed setup. We use a machine with a Xeon Gold 6554S CPU and four ConnectX-7 400 GbE NICs [79] for PXGW. We employ 4 client and 4 server machines, each with a Xeon Gold 5512U CPU and a single ConnectX-7 400 GbE NIC. We configure 1500 B MTU between servers and PXGW, and 9 KB MTU between clients and PXGW. This testbed supports evaluating PXGW at up to 1.6 Tbps of traffic. For tests without MTU translation, we use 1500 B MTU everywhere. We enable TSO and G/LRO on all endpoints. By default, we generate 800 bidirectional TCP flows using iPerf [49] between servers and clients, providing over 1 Gbps per flow. For UDP evaluation, we generate 800 bidirectional UDP flows using iPerf, each generating at least 1 Gbps of traffic.

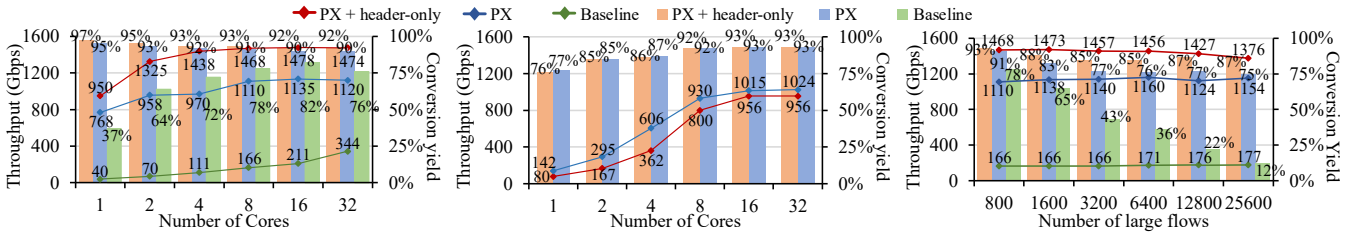
7.1 Performance of PXGW

We evaluate PXGW in terms of throughput, conversion yield, and processing delay. Conversion yield is defined as the fraction of incoming traffic merged into iMTU (9 KB) packets. Since no prior system supports dynamic MTU translation, we implement a baseline using the DPDK GRO library [32] for software packet merging without PX's optimizations.

Throughput and conversion yield. Figure 10a and 10b show the throughput and conversion yield of PXGW with 800 TCP and UDP flows, respectively. Each flow consists of $eMTU_b$ packets to be merged into $iMTU_b$ packets by PXGW. Bars represent conversion yields while lines indicate throughputs. "PX" includes all techniques except header-only DMA, while "PX + header-only" includes the full suite. For TCP traffic, PXGW reaches 1.1 Tbps with 8 cores without header-only DMA, a 6.7× improvement over the baseline. At this point, performance is bottlenecked by DRAM bandwidth, making additional cores ineffective. Enabling header-only DMA alleviates this bottleneck, pushing throughput to 1.47 Tbps. For UDP traffic, PXGW achieves 0.96 Tbps without header-only DMA and 1.01 Tbps with it. UDP performance is lower than TCP due to the lack of hardware offloads like LRO and TSO. Header-only DMA slightly reduces UDP throughput because it consumes additional CPU cycles, and UDP merging is primarily CPU-bound rather than memory-bandwidth-bound. For both protocols, PXGW maintains conversion yield above 90% with more than 8 cores, outperforming the TCP baseline (37%–76%).

Processing delay. We measure the packet processing delay of PXGW from RX to TX. For reference, a saturated 8-core DPDK L3 forwarder exhibits an average processing delay of 1 μs. With 8 cores, PXIO achieves an average processing delay of 17 μs for TCP traffic without header-only DMA and 23 μs with header-only DMA. The slightly higher delay with header-only DMA occurs because the increased throughput allows more flows to use adaptive delayed merging (§4.1), which briefly buffers packets for better merging opportunities. For UDP traffic with 8 cores, the average processing

[9] PDR (Packet Detection Rule), QER (QoS Enforcement Rule), and FAR (Forwarding Action Rule) are per-flow state managed by the UPF.



(a) TCP throughput and conversion yield (b) UDP throughput and conversion yield (c) TCP performance under many large flows

Figure 10: Performance of PXGW and PXGW with header-only DMA

delay is 18 μs regardless of header-only DMA. In contrast, the baseline exhibits 50 μs average processing delay even with 32 cores due to CPU bottlenecks.

Performance with many large flows. Our PXGW prototype targets large flows with approximately 1 Gbps throughput (§2.2). The primary deployment scenario supports up to 1,600 concurrent flows over a 1.6 Tbps aggregate link. To assess PXIO's robustness under more demanding conditions, we evaluate scenarios with higher flow counts. Figure 10c shows the throughput and conversion yield of an 8-core PXGW with varying numbers of concurrent TCP downlink flows. Even with 25,600 flows, PXIO performs effective packet merging by exploiting the burstiness of individual flows. PXIO achieves high scalability by maintaining a minimal per-flow state of only 80 B. With header-only DMA, PXIO achieves 1.34 Tbps while merging 25,600 flows (54 Mbps per flow on average), maintaining 1 μs average processing delay and 87% conversion yield. The conversion yield and delay decrease at high flow counts because flows below 1 Gbps are classified as non-target flows and processed without delay. Despite this, PXIO still achieves a substantially higher yield than the baseline (12% at 25,600 flows).

Ablation study. Figure 11a shows the impact of each PXIO component on an 8-core PXGW with 800 TCP downlink flows. The baseline uses a DPDK GRO-based aggregator. Hardware checksum (HW CSUM) improves throughput from 166 to 215 Gbps and yield from 78% to 81%. LRO/TSO enhances throughput to 686 Gbps (4.1× over baseline) but reduces yield to 72%. Zero-copy (SG DMA) further increases throughput to 1.04 Tbps, but without header-only DMA, it remains limited by DRAM bandwidth with lower yield. PX basic maintains yield above 90% while achieving throughput comparable to SG DMA. Header-only DMA (hdr-only DMA) achieves 1.18 Tbps, overcoming the memory bandwidth limit. Dynamic RXQ scaling (Dyn RXQ) further increases throughput to 1.47 Tbps.

Stability of dynamic RXQ scaling. Figure 11b shows the timeline of RXQ scaling with 8-core PXGW under varying workloads. Initially (①), endpoints launch 80 TCP flows using iPerf with 1 Gbps bandwidth limit. PXGW keeps the RXQ count at 8 to minimize latency, as it is far from saturation. After 4 seconds (②), 800 bandwidth-unconstrained flows are introduced, saturating the existing RXQs. PXGW detects the change and gradually increases the RXQ count. Within 7 seconds, PXGW settles at 128 RXQs, where maximum throughput is achieved. During this transition, a brief throughput dip occurs (1.47 to 1.43 Tbps) as the polling interval increases with more RXQs, but throughput immediately recovers once the queues fill in the next polling cycle. At time slot ③, the 800 flows terminate simultaneously, and PXGW scales RXQs back down to 8. Overall, dynamic RXQ scaling reacts promptly to workload changes with minimal disruption.

Performance with a mix of small flows. We evaluate the robustness of 8-core PXGW under a heavy load of small flows that can hinder MTU translation of large flows. In this test, PXGW is configured to onload large flows identified by their four tuples. We introduce 1M small flows consisting of 96 B packets at various throughput levels. Figure 11c shows that PXGW with flow steering achieves 1 Tbps even with 1M additional small flows at 250 Mpps. In contrast, throughput without flow steering drops to 90 Gbps. PXGW with flow steering maintains 91% conversion yield, whereas the yield drops to 0% without it. This demonstrates that PX effectively filters small flows while maintaining high yield and throughput.

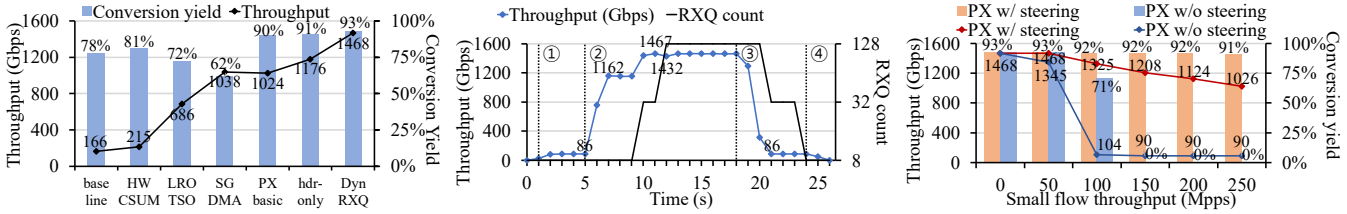
7.2 Performance of NFs and End Nodes

We evaluate the performance improvement of NFs and end nodes when PXGW is deployed.

NFs. We forward traffic through 8-core PXGW to various NFs running on a single core of another machine equipped with a Xeon Gold 6418H CPU and four ConnectX-7 NICs, while varying the iMTU size. Figure 12 shows that throughput scales almost linearly with iMTU size. With 9 KB iMTU, L4LB, NAT, and UPF achieve 478, 147, and 184 Gbps, respectively, outperforming the 1500 B MTU baselines by 4.5×, 4.3×, and 5.1×. These performance gains are achieved without modifying the NFs. Deploying one PXGW improves the performance of all NFs in the b-network.

Sender in a b-network. We evaluate the scenario where PXGW is deployed only in the sender network. We simulate a WAN environment with 10 ms RTT and 0.01% packet loss rate. We run iPerf with a single TCP flow for one minute and measure the average throughput. TCP throughput increases by 2.5× (123 to 303 Mbps) when PXGW uses 9 KB iMTU, even when the receiver is in a network with legacy MTU. This confirms that upgrading only the sender network brings substantial performance benefits. However, the performance improvement is smaller compared to using large MTU on the entire end-to-end path (see Figure 1c). This is because PXGW at the sender segments outgoing 9 KB packets into 1500 B packets, effectively increasing the number of loss events by a factor of six.

Receiver in a b-network. We evaluate the impact of deploying PXGW only in the receiver network. We compare the RX throughput at a receiver with a single CPU core using 9 KB iMTU while fixing eMTU to 1500 B. We measure the maximum throughput by



(a) Ablation study (b) Dynamic RXQ count scaling (c) Performance under heavy small flows

Figure 11: Effectiveness of each technique in PX with header-only DMA

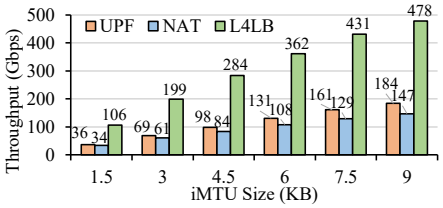


Figure 12: Throughput of NFs

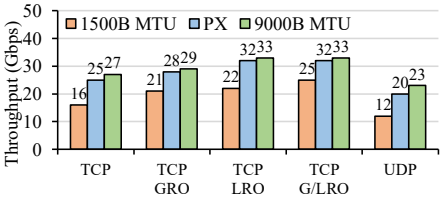


Figure 13: Throughput of a receiver

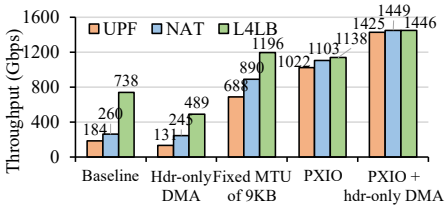


Figure 14: Throughput of PX-ported NFs

injecting multiple 1 Gbps flows until the receiver reaches saturation. We also analyze the impact of LRO and GRO by incrementally enabling these features at the receiver. Figure 13 shows that RX throughput improves by 1.3×–1.6× when using 9 KB iMTU inside the b-network. TCP receivers benefit most from PXGW where offload features such as LRO and GRO are unavailable, such as in mobile devices.[10] We also evaluate the performance of PX-caravan for UDP with the `UDP_GRO` option [53] on. PX-caravan achieves 1.7× better throughput than the baseline. Unlike the sender case, the receiver-side performance benefit is nearly the same as adopting 9 KB MTU on the end-to-end path.

7.3 PX-ported Network Functions

In contrast to the previous experiment where NFs benefit from PXGW translating MTU at the network border, we now evaluate NFs that directly integrate PXIO APIs to perform dynamic packet resizing internally. This approach allows NFs to benefit from PX without deploying a separate PXGW. The idea is to dynamically reduce the number of packets for NF processing by merging or chaining, then split the merged payload by the original MTU size when forwarding to the next hop. We port the same three NFs (UPF, NAT, and L4LB) to use PXIO APIs with fewer than 50 lines of code modification. We turn off adaptive delayed merging (Appendix A) as it is not directly beneficial to NFs.

Figure 14 compares the throughput of PX-ported NFs using 8 cores. PX-ported NFs show performance improvements up to 5.5× over the baseline. Adding header-only DMA further enhances this to up to 7.7×. Header-only DMA alone (or nicmem [86]) performs poorly since it does not reduce the number of packets. PX achieves comparable performance to fixed 9 KB MTU on the entire route. This confirms the benefit of PX and the synergy between PX and header-only DMA.

[10]GRO might be available, but it consumes extra CPU cycles and power.

7.4 Effectiveness of F-PMTUD

We evaluate the effectiveness of F-PMTUD with six CloudLab nodes [80] in the US. We deploy the F-PMTUD daemon on all nodes and probe the PMTU on all possible network paths. We compare the results with Scamper [65], which implements UDP-based PLPMTUD. Both F-PMTUD and Scamper agree on the PMTU values in all paths. The path between Utah and Wisconsin nodes supports 9 KB PMTU, while all others report 1500 B PMTU. We confirm that a 9 KB probe packet sent from the Utah or Wisconsin node to another node is fragmented along the path with the largest fragment size of 1500 B. While both approaches produce identical PMTUs, F-PMTUD is faster as Scamper requires multiple RTTs for convergence. For example, between the Utah and Massachusetts nodes, F-PMTUD takes only 57 ms (1 RTT), which is 368× faster than Scamper.

We now evaluate the large-scale applicability of F-PMTUD by examining whether routers transmit fragmented packets. Using Cloudflare Radar's top 1 million domains, we identify 389,428 active servers with distinct IPs that respond to HTTP requests. To each server, we send two HTTP requests with the same content: a whole-packet request and a fragmented-packet request. We find that 99.98% of the servers respond to the fragmented request, indicating that these servers successfully receive and handle fragmented packets. We further analyze the 59 servers that do not respond to fragmented requests. For 15 servers, sending fragments with reduced TTL produces "ICMP Time Exceeded" messages from their edge routers within the same AS, implying local fragment filtering. The remaining servers provide no useful data as ICMP messages are filtered too early regardless of TTL value. In comparison, the existing ICMP-based PMTUD achieves only 67.0% success rate on the same set of servers. This indicates that F-PMTUD's design, which avoids ICMP reliance, is crucial for effective path MTU discovery.

8 Discussion

Impact on network congestion. A potential concern with incremental MTU upgrades is increased network congestion. If the traffic

sender resides outside the b-network, congestion is unlikely to increase. The sender's rate increases by one packet (MSS) per RTT in TCP, and the MSS remains unchanged (e.g., 1460 B). However, if the sender is in the b-network, a higher send rate with a larger MSS may exacerbate congestion. Recent work [19] indicates that bandwidth contention is infrequent in practice due to improvements in access network bandwidth and per-user bandwidth limitations imposed by ISPs. Nonetheless, if contention occurs, flows with a legacy MTU may receive less bandwidth than their fair share. One mitigation is to dynamically adjust the sender-side MSS between legacy and large MTUs in response to congestion, which requires future investigation.

Scope of F-PMTUD. F-PMTUD assumes that fragmentation behavior is preserved along the path. However, in cellular networks, this assumption breaks down: F-PMTUD probes lose their fragment-size information depending on how packet core functions such as the UPF behave. The UPF reassembles incoming IP fragments and subsequently re-fragments the packet, which can produce fragments of similar size rather than matching its internal MTU. For example, fragments of 1000 B and 800 B may be reassembled and then re-fragmented into two 900 B fragments. This destroys the original fragment-size information that F-PMTUD relies on to infer the path MTU. While cellular networks could be modified to support F-PMTUD, deploying PacketExpress within these networks offers a simpler approach and aligns with their primary use case, as discussed in §2. Similarly, F-PMTUD does not directly apply to IPv6-only paths since IPv6 does not allow fragmentation.

Inter-network coordination. The large-MTU network path can be extended transparently if neighboring networks deploy PX. If a neighboring network supports a compatible iMTU, PXGW can bypass packet size translation and directly forward large-MTU packets across network boundaries. The key challenge is how an autonomous system (AS) can convey its iMTU to neighbor ASes. BGP UPDATE messages already support extensible path attributes, enabling the introduction of a new transitive attribute, AS-MTU, to specify the iMTU an AS can handle (e.g., AS-MTU=9000), as well as PX-CAPABLE, another attribute indicating whether the AS supports PX. This approach leverages existing BGP infrastructure for disseminating iMTU but requires standardization and deployment.

Deployment complexity for PX-caravan. PacketExpress requires endpoint support only when an endpoint directly receives PX-caravans. Since our receiver-side support is implemented at the Linux kernel level, VM-based deployments require updating the corresponding VM images, whereas containerized deployments require no per-container changes because containers share the host kernel. Existing physical middleboxes require no changes as long as they forward packets without inspecting PX-caravan payloads; if they must inspect such payloads, they need to understand the PX-caravan format.

Implementation complexity of PXGW. PXGW is not inherently tied to DPDK. Its core packet-transformation logic—merging incoming packets toward the private network and segmenting outgoing packets back to the next-hop MTU—does not depend on any specific software packet-processing framework. Our DPDK prototype is one implementation that leverages commodity NIC offloads, but the same logic can also run on SmartNICs or other hardware platforms. We have validated this by building an FPGA-based hardware PXGW prototype whose detail is omitted for brevity.

9 Related Work

Effectiveness of large MTU. Several studies have explored the benefits of large packets. Murray et al. [74] observe improved TCP throughput and reduced CPU utilization with large MTUs. Salyers et al. [91] propose an L2 mechanism for efficient forwarding in the core network. Cai et al. [20] analyze the Linux kernel network stack performance in 100 Gbps networks, demonstrating the performance benefits of jumbo frames and highlighting the performance degradation of GRO under high flow counts, similar to our findings. In the context of 5G Advanced, a proposal suggests reducing L2 processing load by concatenating multiple PDCP SDUs [54]. Our research aligns with previous studies on the advantages of large MTUs, but uniquely addresses the incremental deployment of large MTUs in the Internet, which is absent in prior work. This work extends our workshop papers [22, 100] by presenting a complete system design and implementation with a more thorough evaluation. This paper improves the control and data plane of PX with new techniques such as zero-copy packet chaining, header-only DMA, dynamic RXQ scaling, and adaptive delayed merging.

Comparison with L7LB. Hermes [84] shows that L7 load balancers can exploit large MTUs by performing MTU translation at the proxy layer. However, L7LBs must maintain rich per-flow and per-connection state, which complicates scaling. They are also typically connection-terminating proxies that relay data between two TCP connections, incurring overheads such as buffering, reassembly, reliability handling, and frequent memory copies. In contrast, PXGW operates at L4, preserves a single end-to-end flow, and avoids such proxy overheads. It also applies transparently to arbitrary flows rather than being tied to fixed frontend–backend pools.

Ineffectiveness of PMTUD. Path MTU Discovery (PMTUD) has become increasingly unreliable. Custura et al. [25] examine IPv4 and IPv6 PMTUD behavior, finding that only 8.2% of Cisco Umbrella's top 1 million servers successfully perform IPv4 blackhole detection. Their work also reveals a decline in PMTUD success rates over the past eight years. Compared to the 2010 study by Luckie et al. [66], IPv4 success rates drop from 78% to 51%, and IPv6 rates fall from 80% to 30%. These results highlight a trend of networks deprecating support for traditional ICMP-based PMTUD. Unlike conventional PMTUD, F-PMTUD offers reliable path MTU discovery within a single round-trip time.

10 Conclusion

PX presents a practical strategy for reaping the benefits of large MTUs for Internet traffic without changes in external networks. Dynamic MTU translation at PXGW improves the performance of end hosts and middleboxes. PXGW can be implemented as a software-based system using only commodity NIC features. Our evaluation shows that PXGW converts over 90% of 1500 B packets into 9 KB packets while sustaining forwarding throughput exceeding 1 Tbps on an 8-core CPU server. We hope PX becomes a first step toward incremental deployment of large MTUs across the Internet.

Ethics: This work does not raise any ethical issues.

Acknowledgments

We appreciate the insightful feedback and suggestions from SIGCOMM 2026 reviewers. We are especially grateful to Bob Metcalfe, Geoff Thompson, Robert Garner, Dave Redell, and Alan Freier for their valuable discussions on the 1500-byte Ethernet frame size, which helped us identify the authors of the initial Ethernet specification, the "DIX Bluebook" [90]. We also thank Larry Peterson, Jim Kurose, and Matt Mathis for their expert insights on the MTU size and path MTU discovery. This work was supported in part by the ICT Research and Development Program of MSIT/IITP, Korea, under [RS-2024-00349594, Development of networked systems technologies leveraging SmartNIC], [2022-0-00531, Development of in-network computing techniques for efficient execution of AI applications], [RS-2024-00418784, Next-generation Cloud-native Cellular Network Center], [RS-2025-02217106, Ethernet-based high-performance Smart Switch for Data Center], [RS-2026-25507250, Development of a Fabric Platform for Ultra-low Latency, Lossless, and High-bandwidth AI Data Centers], and the New Faculty Startup Fund from Seoul National University. Daehyeok Kim was supported in part by the U.S. National Science Foundation (NSF) Award 2403026. KyoungSoo Park is the corresponding author of this paper.

References

[1] 1982. Internet Protocol (RFC-791). https://datatracker.ietf.org/doc/html/rfc791. Last Accessed: 2026-02-04.
[2] 3GPP. 2015. TS 29.281 General Packet Radio System (GPRS) Tunnelling Protocol User Plane (GTPv1-U). https://www.3gpp.org/DynaReport/29281.htm. Last Accessed: 2026-02-04.
[3] 3GPP. 2016. TS 22.261 Service Requirements for the 5G System (5GS). https://www.3gpp.org/DynaReport/22261.htm. Last Accessed: 2026-02-04.
[4] 3GPP. 2016. TS 29.244 Interface between the Control Plane and the User Plane nodes. https://www.3gpp.org/DynaReport/29244.htm. Last Accessed: 2026-02-04.
[5] 3GPP. 2017. TS 38.323 NR; Packet Data Convergence Protocol (PDCP) specification. https://www.3gpp.org/DynaReport/38323.htm. Last Accessed: 2026-02-04.
[6] Alibaba, Inc. 2013. LVS. https://github.com/alibaba/LVS. Last Accessed: 2026-02-04.
[7] Next G Alliance. 2022. 6G Applications and Use Cases. https://nextgalliance.org/white_papers/6g-applications-and-use-cases/. Last Accessed: 2026-02-04.
[8] Mark Allman. 2003. TCP Congestion Control with Appropriate Byte Counting (ABC) (RFC-3465). https://datatracker.ietf.org/doc/html/rfc3465. Last Accessed: 2026-02-04.
[9] M. Allman, V. Paxson, and W. Stevens. 1999. TCP Congestion Control (RFC-2581. https://datatracker.ietf.org/doc/html/rfc2581. Last Accessed: 2026-02-04.
[10] AMD, Inc. 2024. AMD Pensando 2nd Generation (ELBA) DPU. https://www.amd.com/content/dam/amd/en/documents/pensando-technical-docs/product-briefs/pensando-elba-product-brief.pdf. Last Accessed: 2026-02-04.
[11] Amir, Tariq, and Saeed. 2017. RX and TX Bulking/Batching. https://netdevconf.info/2.1/slides/apr6/network-performance/04-amir-RX_and_TX_bulking_v2.pdf. Last Accessed: 2026-02-04.
[12] Ran Atkinson and Stephen Kent. 1998. Security Architecture for the Internet Protocol (RFC-2401). https://datatracker.ietf.org/doc/html/rfc2401. Last Accessed: 2026-02-04.
[13] Steve Blank. 2018. What the GlobalFoundries' Retreat Really Means. https://spectrum.ieee.org/nanoclast/semiconductors/devices/what-globalfoundries-retreat-really-means. Last Accessed: 2026-02-04.
[14] Ethan Blanton, Dr. Vern Paxson, and Mark Allman. 2009. TCP Congestion Control (RFC-5681). https://datatracker.ietf.org/doc/html/rfc5681. Last Accessed: 2026-02-04.
[15] Ron Bonica, Fred Baker, Geoff Huston, Bob Hinden, Ole Trøan, and Fernando Gont. 2020. IP Fragmentation Considered Fragile (RFC-8900). https://datatracker.ietf.org/doc/html/rfc8900. Last Accessed: 2026-02-04.
[16] Hans-Werner Braun and Yakov Rekhter. 1989. Requirements for Internet Hosts – Communication Layers (RFC-1122). https://datatracker.ietf.org/doc/html/rfc1122. Last Accessed: 2026-02-04.
[17] Broadcom, Inc. 2012. Broadcom BCM5720 Controller Technology. https://docs.broadcom.com/doc/5720-PB01-R. Last Accessed: 2026-02-04.
[18] Broadcom, Inc. 2024. Broadcom BCM57608. https://docs.broadcom.com/doc/BCM57608-PB. Last Accessed: 2026-02-04.
[19] Lloyd Brown, Yash Kothari, Akshay Narayan, Arvind Krishnamurthy, Aurojit Panda, Justine Sherry, and Scott Shenker. 2023. How I Learned to Stop Worrying About CCA Contention. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*.
[20] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*.
[21] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2017. BBR: congestion-based congestion control. *Commun. ACM* 60, 2 (Jan. 2017), 58–66. doi:10.1145/3009824
[22] Youngmin Choi, Junghan Yoon, Younggyoun Moon, and Kyoungsoo Park. 2023. Is Large MTU Beneficial to Cellular Core Networks?. In *Proceedings of the ACM Asia-Pacific Workshop on Networking (APNet)*.
[23] Cisco, Inc. 2022. Cisco Nexus 9800 Series. https://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/nexus9800-series-switches-ds.html. Last Accessed: 2026-02-04.
[24] Cloudflare, Inc. 2024. Cloudflare Radar Domain Ranking. https://radar.cloudflare.com/domains. Last Accessed: 2026-02-04.
[25] Ana Custura, Gorry Fairhurst, and Iain Learmonth. 2018. Exploring Usable Path MTU in the Internet. In *Proceedings of the Network Traffic Measurement and Analysis Conference (TMA)*.
[26] David D. Clark. 1982. Window and Acknowledgement Strategy in TCP (RFC-813). https://datatracker.ietf.org/doc/html/rfc813. Last Accessed: 2026-02-04.
[27] Dr. Steve E. Deering and Bob Hinden. 2017. RTP: Internet Protocol, Version 6 (IPv6) Specification (RFC-8200). https://datatracker.ietf.org/doc/html/rfc8200. Last Accessed: 2026-02-04.
[28] Dr. Steve E. Deering, Jack McCann, and Jeffrey Mogul. 1996. Path MTU Discovery for IP version 6 (RFC-1981). https://datatracker.ietf.org/doc/html/rfc1981. Last Accessed: 2026-02-04.
[29] Dr. Steve E. Deering and Jeffrey Mogul. 1990. Path MTU Discovery (RFC-1191). https://datatracker.ietf.org/doc/html/rfc1191. Last Accessed: 2026-02-04.
[30] Dell, Inc. 2024. Dell PowerSwitch Z-series Spine, Core and Aggregation Switches. https://www.delltechnologies.com/asset/en-us/products/networking/technical-support/dell-powerswitch-z9664f-on-spec-sheet.pdf. Last Accessed: 2026-02-04.
[31] DigiBarn Computer Museum. [n. d.]. Alan O Freier on the D*Machines, the Dolphin, Dorado, Dandelion and more. https://www.digibarn.com/friends/alanfreier/index.html. Last Accessed: 2026-02-04.
[32] DPDK. 2013. DPDK. https://github.com/DPDK/dpdk/blob/main/lib/gro/gro_tcp4.c#L176. Last Accessed: 2026-02-04.
[33] DPDK. 2023. DPDK: Data Plane Development Kit. https://www.dpdk.org/. Last Accessed: 2026-02-04.
[34] DPDK. 2025. 2. Features Overview. https://doc.dpdk.org/guides/nics/features.html#rss-hash. Last Accessed: 2026-02-04.
[35] Gorry Fairhurst, Tom Jones, Michael Tüxen, Irene Ruengeler, and Timo Völker. 2020. Packetization Layer Path MTU Discovery for Datagram Transports (RFC-8899). https://datatracker.ietf.org/doc/html/rfc8899. Last Accessed: 2026-02-04.
[36] Open Networking Foundation. 2022. UPF. https://github.com/omec-project/upf. Last Accessed: 2026-02-04.
[37] Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, and Mohammad Alizadeh. 2016. Juggler: a practical reordering resilient network stack for datacenters. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*.
[38] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan Kostić. 2022. Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*.
[39] Junzhi Gong, Tong Yang, Haowei Zhang, Hao Li, Steve Uhlig, Shigang Chen, Lorna Uden, and Xiaoming Li. 2018. HeavyKeeper: An Accurate Algorithm for Finding Top-k Elephant Flows. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (ATC)*.
[40] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. *SoftNIC: A Software NIC to Augment Hardware*. Technical Report UCB/EECS-2015-155. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html
[41] Hyunwook Hong, Hyunwoo Choi, Dongkwan Kim, Hongil Kim, Byeongdo Hong, Jiseong Noh, and Yongdae Kim. 2017. When Cellular Networks Met IPv6: Security Problems of Middleboxes in IPv6 Cellular Networks. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*.
[42] Intel, Inc. 2017. Intel Ethernet Connection I219. https://www.intel.com/content/www/us/en/content-details/333229/intel-ethernet-connection-i219-product-brief.html. Last Accessed: 2026-02-04.

[43] Intel, Inc. 2018. Intel® Intelligent Fabric Processors. https://www.intel.com/content/www/us/en/products/sku/218648/intel-tofino-2-12-8-tbps-20-stage-4-pipelines/specifications.html. Last Accessed: 2026-02-04.

[44] iQIYI, Inc. 2017. DPVS. https://github.com/iqiyi/dpvs. Last Accessed: 2026-02-04.

[45] Jana Iyengar and Martin Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport (RFC-9000). https://datatracker.ietf.org/doc/html/rfc9000. Last Accessed: 2026-02-04.

[46] Joel Jaeggli. 2019. How big was that? IEPG Meeting @ IETF 105, https://iepg.org/2019-07-21-ietf105/joel-cve-2019-11477.pdf. Last Accessed: 2026-02-04.

[47] Muhammad Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. 2017. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*.

[48] Cullen Fluffy Jennings, John Elwell, and Francois Audet. 2006. Session Initiation Protocol (SIP) URIs for such as Voicemail and Interactive Voice Response (IVR) (RFC-4458). https://datatracker.ietf.org/doc/html/rfc4458. Last Accessed: 2026-02-04.

[49] Dugan Jon, Estabrook John, Ferbuson Jim, Gallatin Andrew, Gates Mark, Gibbs Kevin, Hemminger Stephen, Jones Nathan, Qin Feng, Renker Gerrit, Tirumala Ajay, and Warshavsky Alex. 2021. iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/. Last Accessed: 2026-02-04.

[50] T. Karagiannis, M. Molle, M. Faloutsos, and A. Broido. 2004. A nonstationary Poisson view of Internet traffic. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*.

[51] Georgios P. Katsikas, Tom Barbette, Marco Chiesa, Dejan Kostić, and Gerald Q. Maguire. 2021. What You Need to Know About (Smart) Network Interface Cards. In *Proceedings of the Passive and Active Measurement (PAM)*.

[52] C. A. Kent and J. C. Mogul. 1987. Fragmentation considered harmful. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[53] Michael Kerrisk. 2024. udp(7) — Linux manual page. https://www.man7.org/linux/man-pages/man7/udp.7.html. Last Accessed: 2026-02-04.

[54] Donggun Kim, Sangkyu Baek, Jaehyuk Jang, and Daegyun Kim. 2021. High-Speed Packetization for 5G Advanced. In *Proceddings of the IEEE Globecom Workshops*.

[55] Duckwoo Kim, SeungEon Lee, and KyoungSoo Park. 2020. A Case for SmartNIC-accelerated Private Communication. In *Proceedings of the ACM Asia-Pacific Workshop on Networking (APNet)*.

[56] Masahiko Kitamura, Daisuke Shirai, Kunitake Kaneko, Takahiro Murooka, Tomoko Sawabe, Tatsuya Fujii, and Atsushi Takahara. 2011. Beyond 4K: 8K 60p live video streaming to multiple sites. *Future Gener. Comput. Syst.* 27, 7 (July 2011), 952–959.

[57] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[58] LAN/MAN Standards Committee of the IEEE Standard Society. 2022. *IEEE Standard for Ethernet, Amendment 3: Physical Layer Specifications and Management Parameters for 100 Gb/s, 200 Gb/s, and 400 Gb/s Operation over Optical Fiber Using 100 Gb/s Signaling (IEEE Std 802.3db-2022)*. IEEE.

[59] LAN/MAN Standards Committee of the IEEE Standard Society. 2022. *IEEE Standard for Ethernet (IEEE Std 802.3-2022)*. IEEE.

[60] Larry L. Peterson and Bruce S. Davie. 2021. *Computer Networks: A Systems Approach, 6th Edition*. Morgan Kaufmann.

[61] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. 2020. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science* 368, 6495 (2020), eaam9744. arXiv:https://www.science.org/doi/pdf/10.1126/science.aam9744 doi:10.1126/science.aam9744

[62] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. 1993. On the self-similar nature of Ethernet traffic. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[63] Tony Li, Dino Farinacci, Stanley P. Hanks, David Meyer, and Paul S. Traina. 2000. Generic Routing Encapsulation (GRE) (RFC-2784). https://datatracker.ietf.org/doc/html/rfc2784. Last Accessed: 2026-02-04.

[64] Linus Torvalds. 2008. Linux kernel. https://github.com/torvalds/linux/blob/master/net/ipv4/tcp_offload.c#L270. Last Accessed: 2026-02-04.

[65] Matthew Luckie. 2010. Scamper: a scalable and extensible packet prober for active measurement of the internet. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*.

[66] Matthew Luckie and Ben Stasiewicz. 2010. Measuring path MTU discovery behaviour. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*.

[67] Marek Majkowski. 2015. Path MTU discovery in practice. https://blog.cloudflare.com/path-mtu-discovery-in-practice. Last Accessed: 2026-02-04.

[68] Matt Mathis, Ben Chandler, and John Heffner. 2007. IPv4 Reassembly Errors at High Data Rates (RFC-4963). https://datatracker.ietf.org/doc/html/rfc4963. Last Accessed: 2026-02-04.

[69] Matt Mathis and John Heffner. 2007. Packetization Layer Path MTU Discovery (RFC-4821). https://datatracker.ietf.org/doc/html/rfc4821. Last Accessed: 2026-02-04.

[70] Matt Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis J. Ott. 1997. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *ACM SIGCOMM Computer Communication Review* 27, 3 (1997), 67–82.

[71] Matthew Luckie and Ben Stasiewicz. 2010. Measuring path MTU discovery behaviour. In *Proceedings of the ACM SIGCOMM conference on Internet measurement (IMC)*.

[72] Robert M. Metcalfe and David R. Boggs. 1976. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM* 19, 7 (1976), 395–404.

[73] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[74] David Murray, Terry Koziniec, Kevin Lee, and Michael Dixon. 2012. Large MTUs and Internet Performance. In *Proceedings of the IEEE International Conference on High Performance Switching and Routing*.

[75] Kathleen Nichols and Van Jacobson. 2012. Controlling Queue Delay. *ACM Queue* 55 (05 2012), 42–50. https://queue.acm.org/detail.cfm?id=2209336

[76] NVIDIA, Inc. 2020. NVIDIA ConnectX-5. https://network.nvidia.com/files/doc-2020/pb-connectx-5-en-card.pdf. Last Accessed: 2026-02-04.

[77] NVIDIA, Inc. 2022. NVIDIA ConnectX-6. https://nvdam.widen.net/s/qpszhmhpzt/networking-overal-dpu-datasheet-connectx-6-dx-smartnic-1991450. Last Accessed: 2026-02-04.

[78] NVIDIA, Inc. 2024. NVIDIA CloudXR Suite. https://www.nvidia.com/en-us/design-visualization/solutions/cloud-xr/. Last Accessed: 2026-02-04.

[79] NVIDIA, Inc. 2024. NVIDIA ConnectX-7. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/infiniband-adapters/infiniband-connectx7-data-sheet.pdf. Last Accessed: 2026-02-04.

[80] The University of Utah. 2023. CloudLab. https://cloudlab.us. Last Accessed: 2026-02-04.

[81] Ori, Kam. 2019. DPDK Summit North America, Mountain View CA, November 12-13. https://www.dpdk.org/event/dpdk-summit-na-mountain-view/. Last Accessed: 2026-02-04.

[82] Sergio Orts-Escolano, Christoph Rhemann, Sean Fanello, Wayne Chang, Adarsh Kowdle, Yury Degtyarev, David Kim, Philip L. Davidson, Sameh Khamis, Mingsong Dou, Vladimir Tankovich, Charles Loop, Qin Cai, Philip A. Chou, Sarah Mennicken, Julien Valentin, Vivek Pradeep, Shenlong Wang, Sing Bing Kang, Pushmeet Kohli, Yuliya Lutchyn, Cem Keskin, and Shahram Izadi. 2016. Holoportation: Virtual 3D Teleportation in Real-time. In *Proceedings of the Annual Symposium on User Interface Software and Technology (UIST)*.

[83] Jitendra Padhye, Victor, Firoiu Don, and Towsley Jim Kurose. 1998. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[84] Tian Pan, Enge Song, Yueshang Zuo, Shaokai Zhang, Yang Song, Jiangu Zhao, Wengang Hou, Jianyuan Lu, Xiaoqing Sun, Shize Zhang, Ye Yang, Jiao Zhang, Tao Huang, Biao Lyu, Xing Li, Rong Wen, Zhigang Zong, and Shunmin Zhu. 2025. Hermes: Enhancing Layer-7 Cloud Load Balancers with Userspace-Directed I/O Event Notification. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[85] Chunyi Peng, Chi-yu Li, Guan-Hua Tu, Songwu Lu, and Lixia Zhang. 2012. Mobile data charging: new attacks and countermeasures. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[86] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. 2022. The benefits of general-purpose on-NIC memory. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[87] Boris Pismenny, Adam Morrison, and Dan Tsafrir. 2023. ShRing: Networking with Shared Receive Rings. In *Proceeding of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[88] Boris Pismenny, Adam Morrison, and Dan Tsafrir. 2025. Disentangling the Dual Role of NIC Receive Rings. In *Proceeding of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[89] Thomas H. Ptacek and Timothy N. Newsham. 1998. *Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection*. Technical Report. Secure Networks, Inc. https://insecure.org/stf/secnet_ids/secnet_ids.pdf

[90] Rob Ryan (Intel, Inc.), Rich Seifert (DEC), and Dave Redell (Xerox, Inc.). 1980. *The Ethernet, A Local Area Network, Data Link Layer and Physical Layer Specifications (version 1.0)*.

[91] David Salyers, Yingxin Jiang, Aaron Striegel, and Christian Poellabauer. 2007. JumboGen: Dynamic Jumbo Frame Generation for Network Performance Scalability. *SIGCOMM Computer Communication Review* 37, 5 (oct 2007), 53–64. doi:10.1145/1290168.1290174

[92] Samsung Research, Inc. 2020. The Next Hyper-Connected Experience for All. https://cdn.codeground.org/nsr/downloads/researchareas/6G%20Vision.pdf. Last Accessed: 2026-02-04.

[93] GyÖrgy Terdik and Tibor Gyires. 2009. LÉvy Flights and Fractal Modeling of Internet Traffic. *IEEE/ACM Transactions on Networking* 17, 1 (2009), 120–129.

[94] The kernel development community. 2024. The Linux Kernel documentation. https://docs.kernel.org/networking/ip-sysctl.html. Last Accessed: 2026-02-04.

[95] Martin Thomson and Sean Turner. 2021. Using TLS to Secure QUIC (RFC-9001). https://datatracker.ietf.org/doc/html/rfc9001. Last Accessed: 2026-02-04.

[96] Junfeng Wang, Hongxia Zhou, Fanjiang Xu, and Lei Li. 2005. Hyper-Erlang Based Model for Network Traffic Approximation. In *Proceedings of the Parallel and Distributed Processing and Applications ISPA*.

[97] Yanshu Wang, Dan Li, Yuanwei Lu, Jianping Wu, Hua Shao, and Yutian Wang. 2022. Elixir: A High-performance and Low-cost Approach to Managing Hardware/Software Hybrid Flow Tables Considering Flow Burstiness. In *1Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*.

[98] Hao Wu, Hsu-Chun Hsiao, and Yih-Chun Hu. 2014. Efficient Large Flow Detection over Arbitrary Windows: An Algorithm Exact Outside an Ambiguity Region. In *Proceedings of the ACM SIGCOMM conference on Internet measurement (IMC)*.

[99] Junghan Yoon. 2026. PacketExpress. https://github.com/TNET-SNU/PacketExpress. Last Accessed: 2026-07-03.

[100] Junghan Yoon, Youngmin Choi, Juyoung Park, Daehyeok Kim, Changhoon Kim, and KyoungSoo Park. 2025. Towards Incremental MTU Upgrade for the Internet. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*.

[101] Wenxiao Zhang, Feng Qian, Bo Han, and Pan Hui. 2021. DeepVista: 16K Panoramic Cinema on Your Mobile Device. In *Proceedings of the Web Conference*.

[102] Sihao Zhao, Hatem Abou-zeid, Ramy Atawia, Yoga Suhas Kuruba Manjunath, Akram Bin Sediq, and Xiao-Ping Zhang. 2021. Virtual Reality Gaming on the Cloud: A Reality Check. https://arxiv.org/abs/2109.10114. Last Accessed: 2026-02-04.

[103] Glen Zorn, Gurdeep-Singh Pall, and Kory Hamzeh. 1999. Point-to-Point Tunneling Protocol (PPTP) (RFC-2637). https://datatracker.ietf.org/doc/html/rfc2637. Last Accessed: 2026-02-04.

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

APPENDIX A. Adaptive Delayed Merging

As introduced in §4.1, PXIO employs adaptive delayed merging to maximize packet merging efficiency while controlling latency. PXGW reads multiple packets as a batch from a NIC and merges them into larger packets, capping their size at the target iMTU. However, blindly repeating this process often results in suboptimal merging performance, producing many packets smaller than iMTU. To address this issue, PXIO adopts a strategy inspired by TCP delayed ACK [16, 26]: instead of immediately forwarding incomplete packets, PXIO optimistically waits for upcoming packets to arrive for merging. Longer waits enable better merges, yet they can increase end-to-end latency and reduce throughput. To strike the right balance, PXIO predicts the arrival time of subsequent packets and uses it as the maximum delay window for each flow.

Variance factors in inter-arrival time. Modern TCP senders adopt packet pacing to regulate traffic [21, 73, 75], but traffic burstiness at receivers still persists due to multi-layered factors. First, at the application level, data is often issued in chunks, establishing an inherent *on-off* pattern that TCP inherits [62]. Second, at the driver and hardware levels, TSO/GSO can generate many packets back-to-back, while doorbell batching [11] allows submitting multiple packet descriptors simultaneously. Third, even with flow pacing at the source, the arrival pattern can be highly stochastic as packets are interleaved with unpredictable cross-traffic along the Internet path. These factors apply to UDP as well.

Stochastic abstraction. Instead of attempting to model each factor of delay—which is often infeasible due to its non-deterministic interactions—we abstract away these uncertainties by treating them as a sequence of independent stochastic events. By assuming a Poisson arrival process for Internet traffic [50], we can model the total waiting time as a conservative bound. This stochastic approach allows PXIO to be robust against variance factors without needing to inspect the underlying cause of each delay. Under this abstraction, the inter-arrival time (IAT) for each flow follows an exponential distribution, and we can derive a global average IAT from the flow's throughput R.

PXIO utilizes the Gamma distribution [93, 96] to predict the cumulative time needed for k packets to achieve iMTU, modeling the delay as a series of k independent, exponentially-distributed events. The following Equation (1) expresses the Estimated Time of Merge (ETM):

$$ETM = \Gamma^{-1}\left(p = 0.95;\ k = \left\lceil \frac{\text{iMTU} - L}{\text{PMTU}} \right\rceil,\ \theta = \frac{\text{PMTU}}{R}\right) \quad (1)$$

where k is the estimated number of additional packets required to reach the target iMTU; R is the throughput; L is the merged size so far; and $\Gamma^{-1}(p;\ k, \theta)$ is the p-quantile of a Gamma distribution with shape k and scale θ.

Avoiding unnecessary delay. To avoid excessive delay windows, PXIO enables adaptive delayed merging only for flows exceeding 1 Gbps; slower flows undergo only stateless merging. PXIO also bypasses flows whose conversion yield already exceeds 93%, since further waiting provides little additional merging benefit. To avoid oscillation between enabling and disabling delayed merging, PXIO tracks conversion yield in real time rather than evaluating it only at fixed intervals.

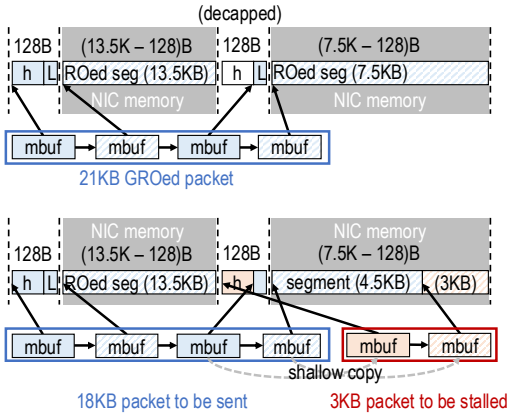


Figure 15: Selectively stashing partial segment without copy

Avoiding head-of-line blocking. To preserve the packet order and minimize head-of-line (HoL) blocking, adaptive delayed merging is applied only to the tail of the "item" queue. In PXIO, an "item" represents an entry in the flow's merged packet list, ordered by increasing TCP sequence numbers for TCP flows and in FIFO order for UDP flows. The presence of multiple items in the queue typically indicates that packet loss or out-of-order delivery has occurred. In such scenarios, waiting for middle packets incurs HoL blocking. Thus, PXIO only applies the ETM-based delay when the tail item is incomplete.

Partial stashing. If a leftover packet falls within its delay window, PXIO places it into a "stash" area for potential merging. For instance, assume PXGW receives two LRO-merged packets of 13.5 KB and 7.5 KB chained by packet merging, with iMTU set to 9 KB. Before transmitting them using scatter-gather DMA with TSO, PXIO trims the 7.5 KB packet to 4.5 KB and shallow-copies the remaining 3 KB into the stash. Since shallow copying only increments the buffer's reference count, no payload is physically copied. Figure 15 illustrates the overview of partial stashing for this case.