WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

High IO Performance User Space Networking with eBPF/XDP and AF_XDP: Design, Tuning, and Architectural Insights

Manish Kumar*

AMD, India

DOI : https://doi.org/10.55248/gengpi.6.0825.3072

A B S T R A C T

The growth of edge computing, 5G deployments, and high IO services has intensified the demand for packet processing methods that combine raw throughput with operational stability. Traditional kernel-bypass approaches can deliver speed but often sacrifice integration with Linux's networking ecosystem. This paper presents eBPF/XDP with AF_XDP zero-copy mode in Linux 6.4 as a high-performance, kernel-integrated alternative. We outline its architecture, show how it eliminates key bottlenecks in the standard kernel data path, and detail proven tuning strategies for achieving line-rate user space I/O. Implementation examples are provided from both kernel and user space perspectives, supported by performance-oriented design practices. eBPF/XDP is a methodology for building low-latency and high-throughput applications.

Keywords: eBPF, XDP, AF_XDP, Zero-Copy Networking, Fast Path Packet Processing, High-Performance User Space I/O, Low-Latency Packet Processing, 5G

1. Introduction

Over the past decade, the boundaries of high-speed networking have shifted. The ability to process packets at tens of millions per second is no longer confined to carrier-grade routers or specialized hardware accelerators. Today, data centers, content delivery networks, IoT gateways, and 5G core nodes all face the same challenge: how to move data from the network interface to application logic quickly enough to meet ever-tighter latency budgets without breaking the operational model of a Linux-based system.

Traditional approaches to extreme packet rates have relied on kernel-bypass frameworks. These remove the operating system's networking stack from the data path, giving applications direct control of the NIC's DMA buffers. While this delivers raw speed, it often comes at a cost: the loss of kernel-level safety checks, routing capabilities, and native observability tools. This trade-off makes such solutions harder to integrate into dynamic, multi-tenant, or fast-changing production environments.

eBPF (extended Berkeley Packet Filter) with XDP (Express Data Path) offers a different path forward. By attaching a small, verifiable program directly to the NIC receive path inside the kernel, XDP can filter, modify, or redirect packets before the usual network stack processing begins. When paired with AF_XDP sockets in zero-copy mode, those packets can be delivered straight into pre-allocated user space memory—bypassing socket buffer allocations and eliminating unnecessary copies—while still operating under the safeguards and integration of the Linux kernel.

This paper examines how this architecture works, why it outperforms the normal kernel path for user space I/O, and how, when tuned correctly and deployed on modern platforms such as AMD Zen 5, it can sustain line-rate performance for the most demanding workloads in edge computing, AI, and 5G environments.

2. eBPF/XDP and AF_XDP – Architectural Overview

2.1 XDP Fast Path Execution

XDP runs at the lowest point in the kernel's network receive path, directly after the NIC driver has populated a receive descriptor but before any allocation of socket buffers (SKBs). At this point, the kernel has minimal metadata overhead, and processing can:

Drop unwanted packets immediately, reducing CPU waste.

Modify packet headers in-place

Redirect packets to another interface or to an AF_XDP socket for user space processing.

* Corresponding author: E-mail address: manishks99@outlook.com

2.2 AF_XDP Socket Model

AF_XDP uses a UMEM, a contiguous, page-aligned shared memory region which both kernel and user space can access without copies. This UMEM is divided into fixed size frames and manages four lockless rings:

Fill Queue (FQ): User space posts empty frames for the kernel to fill.

RX Ring: The kernel posts received packet frames to user space.

TX Ring: User space posts frames to send back to the kernel/NIC.

Completion Queue (CQ): Kernel notifies user space when TX frames are sent and can be reused.

2.3 Zero-Copy Mode

UMEM buffers are directly written by NIC DMA engines when operating in XDP_ZEROCOPY mode. No intermediate copies occur, eliminating the largest per-packet overhead in traditional kernel networking. When zero-copy is not supported, XDP_COPY mode is available, but with slightly higher CPU cost.

2.4 Safety and Programmability

XDP applications should meet following safety criteria:

No unsafe memory access.

Bounded loops with deterministic execution.

Restricted helper calls.

This model allows safe deployment of programmable packet logic at driver level (Høiland-Jørgensen et al., 2018).

3. XDP vs Normal Kernel Path for User Space I/O

3.1 Normal Linux kernel path

Packets are received by the NIC and placed in a DMA buffer.

The kernel allocates an SKB and copies packet data into it.

Protocol stack processing (Ethernet, IP, TCP/UDP) is performed.

The packet may be queued for user space delivery via sockets, involving additional copies and scheduling overhead.

These steps introduce multiple memory allocations, data copies, and context switches, each adding latency and consuming CPU cycles.

3.2 XDP + AF_XDP Path

SKB allocation is entirely bypassed.

The packet remains in a preallocated UMEM buffer shared between kernel and user space.

No data copy is needed (in zero-copy mode).

Protocol processing is deferred to user space only if required. This reduction in memory operations and kernel scheduling points is the primary reason AF_XDP applications can process tens of millions of packets per second in software, especially when paired with NUMA locality and batch processing.

4. Comparative Code Examples: Normal Linux Kernel vs eBPF/XDP AF_XDP Packet Processing

Packet processing paths differ significantly between the traditional Linux networking stack and an AF_XDP fast-path. In the kernel path, packets traverse multiple layers of protocol handling before reaching user space. With eBPF/XDP and AF_XDP, packets can be processed directly at the driver level, bypassing much of the kernel network stack. The code snippet shown here is for reference only.

4.1 Rx Path in the Normal Linux Kernel

// Simplified kernel RX path

```
static int driver_poll(struct napi_struct *napi, int budget)
{
   struct sk_buff *skb;
   int work_done = 0;

   while (work_done < budget && (skb = driver_fetch_packet())) {
      // Pass packet to kernel stack
      netif_receive_skb(skb);  // Goes through protocol layers (L2->L3->L4)
      work_done++;
   }

   if (work_done < budget)
      napi_complete_done(napi, work_done);

   return work_done;
}
```

The NIC driver fetches packets from the DMA ring into sk_buff structures.

Each packet is passed through netif_receive_skb() into the full kernel networking stack.

The stack performs Ethernet, IP, and TCP/UDP processing before the payload reaches a socket in user space.

This involves multiple memory copies and context switches, adding latency.

4.2 Tx Path in the Normal Linux Kernel

```
// User writes data -> Kernel socket send path
ssize_t send(int sockfd, const void *buf, size_t len, int flags)
{
   // Copy from user space to sk_buff
   skb = sock_alloc_send_skb();
   memcpy(skb_put(skb, len), buf, len);

   // Queue packet to NIC driver
   dev_queue_xmit(skb);
}
```

Data is copied from user space to kernel space before being queued for transmission.

Protocol headers are added in the kernel, and the packet goes through qdisc scheduling before DMA.

4.3 Rx Path with eBPF/XDP and AF_XDP

```
// XDP program at driver level
```

```
SEC("xdp")
int xdp_rx_redirect(struct xdp_md *ctx)
{
    // Redirect to AF_XDP socket
    return bpf_redirect_map(&xsks_map, ctx->rx_queue_index, 0);
}

// User space AF_XDP application
while (running) {
    // Poll RX ring for packets
    if (xsk_ring_cons__peek(&rx, BATCH_SIZE, &idx_rx)) {
        for (i = 0; i < BATCH_SIZE; i++) {
            void *pkt = xsk_umem__get_data(umem_buffer, *xsk_ring_cons__rx_desc(&rx, idx_rx + i)->addr);
            process_packet(pkt); // Application logic
        }
        xsk_ring_cons__release(&rx, BATCH_SIZE);
    }
}
```

The XDP program runs at the earliest driver hook, before packet allocation into sk_buff.

Using bpf_redirect_map(), packets are sent directly into an AF_XDP socket bound to a UMEM region in user space.

The user space process reads packet data directly from DMA buffers (zero-copy mode), avoiding additional memory copies.

Processing latency is reduced by eliminating the kernel's protocol stack traversal.

4.4 Tx Path with eBPF/XDP and AF_XDP

```
// User space AF_XDP transmit loop
while (running) {
    if (xsk_ring_prod__reserve(&tx, BATCH_SIZE, &idx_tx) == BATCH_SIZE) {
        for (i = 0; i < BATCH_SIZE; i++) {
            struct xdp_desc *desc = xsk_ring_prod__tx_desc(&tx, idx_tx + i);
            desc->addr = pkt_buffer_addr[i];
            desc->len = pkt_length[i];
        }
        xsk_ring_prod__submit(&tx, BATCH_SIZE);
    }

    sendto(xsk_socket__fd(xsk), NULL, 0, MSG_DONTWAIT, NULL, 0); // Kick Tx
}
```

Packets are written directly into UMEM buffers from the application.

Transmission is initiated without kernel socket handling or qdisc scheduling.

The NIC's DMA engine fetches buffers directly from UMEM.

Table 1 – Key Performance Differences

Feature	Normal Kernel Path	eBPF/XDP AF_XDP Path
Memory Copies	Multiple (user→kernel, kernel buffers)	Zero-copy possible
Context Switches	Frequent	Minimal (polling user space loop)
Protocol Stack	Full L2–L4	Optional (can be bypassed)
Latency	Higher	Significantly lower
Throughput	Limited by stack overhead	NIC line-rate achievable

5. Best Practices for High-Spec User Space Packet Processing

High-performance AF_XDP applications are highly sensitive to small inefficiencies in hardware setup, memory layout, CPU scheduling, and application code paths. Achieving and sustaining line-rate throughput requires a deliberate approach to each subsystem. Below, each best-practice area is expanded with practical considerations, rationale, and real-world tuning tips.

5.1 Hardware and NIC Setup

5.1.1 Driver and Firmware Compatibility

AF_XDP zero-copy mode depends on NIC driver support. Not all drivers provide direct DMA into UMEM buffers, and some require specific firmware revisions to function correctly. Intel and Mellanox recent NICs are supported. In practice, upgrading NIC firmware has solved unexplained drop patterns and queue starvation issues in multiple deployments.

5.1.2 Receive Side Scaling (RSS) and Queue/Core Affinity

By default, NICs distribute incoming packets across multiple queues using a hash of packet headers. AF_XDP benefits from fixing this mapping so that each queue is handled by a dedicated CPU core hence avoiding cache line thrashing when packet moves between the cores. On Linux, ethtool -X can set explicit RSS mappings to ensure predictable core assignment.

5.1.3Interrupt Moderation and Polling Strategy

NICs moderate interrupts to batch work and reduce CPU overhead, but at the cost of latency. AF_XDP applications that poll the RX ring directly can reduce or disable moderation altogether, relying instead on busy-polling loops. This is essential for sub-10 µs packet turnaround in low-latency trading or real-time control systems.

5.1.4 Offload Configuration

Offload configurations help in general-purpose networking, they distort the packet stream in fast-path applications. Disabling them ensures AF_XDP sees each packet in its original form, improving predictability and matching application-level MTU assumptions.

5.2 Memory and NUMA Optimization

5.2.1 NUMA Locality and Latency Reduction

In multi-socket servers, a NIC is attached to a specific NUMA node. Allocating UMEM from the same node prevents cross-node memory access, which can add tens of nanoseconds per packet. While that seems small, at 20 Mpps the cumulative penalty can be substantial. Following usage can lead to memory locality

- numactl
- cpunodebind
- membind

5.2.2 UMEM Sizing for Bursty Traffic

Once UMEM limit is breached, packet drop starts happening. The UMEM must have enough frames to hold in-flight packets until the application recycles them. The following formula can be used to calculate minimum UMEM buffers - (link_speed_in_bps / 8) * burst_time_in_seconds / frame_size

For a 100 Gbps NIC and 64-byte frames over a 1 ms burst, this can mean thousands of frames per queue.

5.2.3 Frame Size Alignment

Frames must be large enough to hold the maximum packet plus metadata padding. Misaligned or undersized frames cause split-packet handling in the NIC, which wastes DMA cycles and reduces throughput. For Ethernet MTU 1500, a 2048-byte frame size is standard, ensuring alignment to 2 KB boundaries.

5.3 Socket Configuration

5.3.1 Zero-Copy Activation

When binding the AF_XDP socket, explicitly request XDP_ZEROCOPY. Without this, the kernel defaults to copy mode, adding an unnecessary memory transfer per packet. Copy mode is acceptable for NICs without zero-copy support but is measurably slower, often halving throughput in small-packet tests.

5.3.2 XDP_USE_NEED_WAKEUP Flag

This flag optimizes wakeup behavior in polling applications. Without it, the kernel might issue unnecessary wakeups when rings are already full or empty, wasting CPU cycles. When enabled, wakeups occur only when truly needed, improving CPU efficiency in high-load scenarios.

5.3.3 Shared UMEM in Multi-Queue Apps

When an application handles multiple queues, using XDP_SHARED_UMEM allows all queues to share a single UMEM allocation. This reduces memory footprint and improves cache reuse for packet metadata, but requires careful frame lifecycle management to avoid reuse conflicts.

5.4 Application Design

5.4.1 Batch-Oriented Processing

The cost of reading a single packet from the RX ring and returning it to the kernel is high when done one-by-one. Batching reading 32 to 64 descriptors at a time amortizes the ring access and syscall overhead. This can yield 20–30% throughput gains without affecting latency significantly.

5.4.2 Cache Prefetching for Predictable Latency

Before accessing packet data, prefetching the relevant cache lines with __builtin_prefetch() hides DRAM access latency. In small-packet processing, this reduces stall cycles, making the RX loop more predictable under high load.

5.4.3 Lock Avoidance and Thread-Per-Queue Design

Shared locks across threads cause cache line bouncing, which is particularly harmful in packet processing loops. The most effective design is one thread pinned to one RX/TX queue pair, eliminating cross-thread contention entirely.

5.5 CPU Scheduling and Power Management

5.5.1 Dedicated Core Allocation

The Linux scheduler will move threads between cores, to avoid this thread should be pinned explicitly. This disrupts CPU cache warmth and increases scheduling latency. Using sched_setaffinity() or isolating cores at boot (isolcpus) guarantees steady performance.

5.5.2 Disabling Dynamic Frequency Scaling

The default CPU governor may scale core frequency based on load, introducing unpredictable latency. Setting the governor to performance ensures the CPU runs at full clock speed at all times.

5.5.3 Hyper-Threading Considerations

Hyper-threading shares execution units between logical cores. For compute-heavy packet processing, disabling hyper-threading for AF_XDP threads avoids competition for execution resources and improves worst-case latency.

5.6 Monitoring and Diagnostics

5.6.1 NIC Hardware Counters

ethtool -S exposes per-queue and per-port statistics such as drops, CRC errors, and DMA failures. If Rx drops are observed it might be due to queue underprovisioning or missed polling cycles.

5.6.2 AF_XDP Ring Instrumentation

The fill, RX, TX, and completion rings can be monitored via BPF maps to measure headroom and backlog in real time. A consistently full RX ring signals that the application is not consuming packets fast enough.

5.6.3 Latency Profiling Without Disturbance

Integrating eBPF perf events allows timestamping packets at RX and TX without adding excessive overhead. This supports ongoing latency monitoring in production without requiring packet duplication.

5.6.4 Sustained Load Validation

Synthetic traffic generators (pktgen, Trex) or hardware testers can validate that an application runs stably at expected rates for hours or days. This testing often reveals subtle memory leaks or queue imbalance issues not seen in short tests.

5.6.5 Proactive Bottleneck Detection

Tracking queue occupancy trends over time can expose performance degradation before packet loss occurs. For example, rising CQ backlog can indicate that the TX path is saturating before the RX path, suggesting NIC transmit pacing or application send logic needs tuning.

6. Leveraging AMD Zen 5 Architecture for High-Performance User Space I/O

The AMD Zen 5 microarchitecture enhanced instruction pipeline, execution bandwidth, and memory hierarchy while integrating DDR5 and PCIe 5.0 support. Hence creating an ideal foundation for accelerating eBPF/XDP workloads paired with AF_XDP. High IO workloads demand predictable low latency, high throughput, and NUMA-aware scaling, align closely with Zen 5's design objectives.

6.1 Frontend Enhancements and Branch Prediction

Zen 5 doubles the instruction fetch rate to two 32-byte instructions per cycle and adds a dual-branch prediction engine capable of a two-ahead lookahead. This reduces pipeline stalls caused by conditional packet parsing and routing logic in tight polling loops, ensuring steady packet throughput.

6.2. Expanded Cache and Memory Subsystem

Each Zen 5 core includes a 48 KB L1 data cache, 1 MB L2 cache, and CCD-shared 32 MB L3 cache. The I/O die supports DDR5 and PCIe 5.0 for higher NIC DMA throughput. Larger caches reduce DRAM fetches for UMEM metadata, and PCIe 5.0 ensures the NIC is never bandwidth-bound, even at 200 Gbps.

6..3 AVX-512 and SIMD Improvements

Zen 5 introduces full-width AVX-512 execution with six vector pipelines. Vectorized packet classification and checksum computations can be processed in half the cycles compared to AVX2, improving batch-processing efficiency in AF_XDP applications.

6.4 NUMA and CCD Locality

The CCD-based layout ensures that each group of cores has direct access to its local L3 cache and memory controllers via Infinity Fabric. Bind threads and buffer allocations to the same CCD as the NIC's NUMA node to avoid cross-CCD latency penalties.

6.5 SMT and Core Utilization Strategy

While Zen 5 supports Simultaneous Multi-Threading (SMT), AMD's tuning guide notes that disabling SMT for latency-sensitive workloads can improve predictability by eliminating shared execution-unit contention. Align all workload thread in a single CCD to avoid cross CCD latency resulting in better IO performance.

7. Performance Uplift in AI, Edge, and 5G Deployments on AMD Platforms with eBPF/XDP

Many AI, Edge computing, and 5G deployments share the same technical challenge: moving large volumes of data from network interfaces to the CPU and back with minimal latency and CPU overhead. The eBPF/XDP framework, combined with AF_XDP, enables this by allowing applications to process packets at the earliest point in the Linux networking stack — even before socket-level handling — and deliver them directly to user space without kernel context-switching or extra copies.

On AMD Zen 5 EPYC platforms, this combination is especially powerful because Zen 5's architectural strengths — wide execution pipelines, large caches, high DDR5 bandwidth, and PCIe 5.0 I/O — align perfectly with the requirements of high-speed, low-jitter packet processing.

7.1 AI Inference at the Edge

In AI inference gateways, large sensor or video datasets are often streamed over Ethernet to be preprocessed before passing to accelerators or CPU-based inference engines.

Role of eBPF/XDP: AF_XDP enables direct zero-copy delivery of packetized data from the NIC into user space AI pipelines, bypassing the TCP/IP stack overhead.

Zen 5 Advantage: AVX-512 vector units process batches of input data (e.g., image frames, audio samples) more efficiently, while DDR5 memory quickly feeds model weights and activation buffers.

Result: Lower latency from NIC to inference, enabling real-time classification or decision-making at the edge.

7.2 Edge Computing Platforms

Edge platforms running microservices, such as security scanning, real-time video analysis, or telemetry data aggregation. It must handle sustained high packet rates while supporting a variety of concurrent workloads.

Role of eBPF/XDP: Packet classification and filtering can be done in line at the XDP hook, letting only relevant traffic reach user space containers, reducing CPU load for non-critical flows.

Zen 5 Advantage: Its large L3 cache helps keep AF_XDP ring buffers and UMEM data resident in cache, while NUMA-aware scheduling minimize delay in Inter CCD communications.

Result: Higher throughput per watt and predictable performance even when workloads are mixed.

7.3 5G RAN Deployments

RAN processing (e.g., fronthaul eCPRI traffic) is extremely latency-sensitive, often with budgets in hundreds of microseconds.

Role of eBPF/XDP: AF_XDP can handle fronthaul packet reception in deterministic, lock-free polling loops, delivering data to Layer 1/2 software stacks without socket overhead.

Zen 5 Advantage: High IPC, and low cache latency will help in achieving 3gpp sub-frame timing requirements.

Result: Stable fronthaul performance without packet jitter under high load.

7.4 5G UPF in the Core Network

The User Plane Function (UPF) processes millions of GTP-U packets per second for subscriber traffic.

Role of eBPF/XDP: GTP-U tunnel parsing and fast-path routing can be offloaded to in-kernel XDP programs, with AF_XDP delivering only relevant payloads to user space, cutting per-packet processing time.

Zen 5 Advantage: AVX-512 enables batch header parsing; PCIe 5.0 and DDR5 sustain multiple NICs in parallel without bottlenecks.

Result: UPF nodes achieve line-rate throughput while freeing cores for control-plane workloads.

5. Conclusion

The work presented here shows that eBPF/XDP, when combined with AF_XDP in zero-copy mode, can deliver packet I/O performance once thought possible only with specialized hardware or kernel-bypass stacks. By intercepting traffic as it arrives at the NIC driver, XDP removes costly steps like socket buffer allocation, deep protocol parsing, and repeated memory copying. AF_XDP then provides a direct, lock-free path into user space, allowing applications to operate at line rate with predictable latency.

What makes this approach stand out is that it does not abandon the Linux kernel's strengths. Safety checks, observability tools, and protocol integration remain intact, so the fast path can be used in production without losing operational control. When deployed on hardware designed for high-throughput, low-latency work—such as AMD's Zen 5 EPYC platforms—the benefits compound: wide execution pipelines, large caches, and high-bandwidth I/O help sustain performance even under demanding AI, edge, and telecom workloads.

In short, eBPF/XDP with AF_XDP offers a rare balance: the speed of a bypass solution, the safety of the kernel, and the flexibility to adapt to future network demands. For engineers building the next generation of low-latency, high-capacity systems, it is not just an alternative path—it is the path forward.

By executing at the NIC driver's ingress point, eBPF/XDP bypasses the most expensive stages of the Linux networking stack for user space delivery. When combined with AF_XDP in zero-copy mode and configured following proven optimization techniques, it delivers a reliable framework for high-performance packet processing in production environments. In contrast to kernel-bypass solutions, it preserves the kernel's built-in safety mechanisms, monitoring capabilities, and networking features, making it well-suited for the dynamic demands of edge, cloud, and telecommunications workloads.

References

T. Høiland-Jørgensen, J. D. Brouer, and D. Borkmann, "The Express Data Path: Fast programmable packet processing in the operating system kernel," Proceedings of the 2018 ACM SIGCOMM Conference, ACM, 2018. doi:10.1145/3230543.3230554

J. D. Brouer and T. Høiland-Jørgensen, "XDP: Challenges and future work," Linux Plumbers Conference, 2018. [Online]. Available: https://lpc.events

M. A. M. Vieira, M. S. Castanho, and R. D. G. Pacífico, "Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications," ACM SIGCOMM Computer Communication Review, vol. 50, no. 5, pp. 62–69, 2020. doi:10.1145/3432920.3437301

M. Abranches, E. Hunhoff, and R. Eswara, "LinuxFP: Transparently accelerating Linux networking," IEEE International Conference on High Performance Switching and Routing (HPSR), 2024. doi:10.1109/HPSR60514.2024

Linux Kernel Documentation: AF_XDP – Address Family for High Performance Packet Processing, kernel.org, 2023. [Online]. Available: https://www.kernel.org/doc/html/v6.4/networking/af_xdp.html

AMD, 5th Gen AMD EPYC™ Processor Architecture White Paper. [Online]. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/5th-gen-amd-epyc-processor-architecture-white-paper.pdf

AMD, AMD EPYC™ 9004 Series Processors Architecture Overview & Tuning Guide. [Online]. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/tuning-guides/58015-epyc-9004-tg-architecture-overview.pdf

Linux Networking Documentation: XDP and eBPF, kernel.org, 2023. [Online]. Available: https://www.kernel.org/doc/html/latest/networking/xdp.html

Cilium Project, eBPF and XDP Reference Guide, 2023. [Online]. Available: https://cilium.io

F. Hauser et al., "Accelerating Linux networking with eBPF offload," USENIX Annual Technical Conference (ATC), 2021.