FLASH: Fast Linked AF_XDP Sockets for High Performance Network Function Chains

Debojeet Das, Kevin Prafull Baua, Aditya Kansara, Arghyadip Chakraborty
Dheeraj Kurukunda, Mythili Vutukuru, Purushottam Kulkarni
Indian Institute of Technology Bombay
Mumbai, India

Abstract

Recent advances in the Linux kernel, such as eXpress Data Path (XDP) and AF_XDP sockets, enable high-speed packet processing for software NFs while preserving access to kernel features. However, the default AF_XDP implementation in the Linux kernel does not permit easy and performant NF chaining, e.g., zero-copy transfer of packets across NFs co-located on the same host. While prior work has proposed solutions for optimized NF chaining in the context of kernel bypass frameworks like DPDK that operate entirely in userspace, such solutions do not extend easily to AF_XDP, because the AF_XDP datapath is fragmented across the kernel driver and userspace. This paper introduces FLASH, a low-overhead in-kernel chaining mechanism for AF_XDP sockets. FLASH enables zero-copy packet transfers for FLASH-native NFs, and single-copy packet transfers for legacy AF_XDP NFs. Further, via integration with K8s, FLASH supports the deployment of unprivileged containerized NFs on cloud platforms. Our work contributes several novel modifications to the AF_XDP datapath in the kernel to implement optimized NF chaining, and provides userspace libraries/APIs to easily build NFs that leverage FLASH. Our evaluations show that FLASH matches the performance of userspace DPDK-based NF chaining frameworks, while outperforming the best available AF_XDP-based alternatives by up to 2.5× in throughput, and achieving the lowest latency among all NF chaining frameworks.

CCS Concepts

• **Networks** → **Cloud computing**; **Programmable networks**; • **Software and its engineering** → **Operating systems**.

Keywords

Network functions, Shared memory, Multi-tenant, Cloud computing, XDP, AF_XDP, Kernel-Bypass

ACM Reference Format:
Debojeet Das, Kevin Prafull Baua, Aditya Kansara, Arghyadip Chakraborty, Dheeraj Kurukunda, Mythili Vutukuru, Purushottam Kulkarni. 2025. FLASH: Fast Linked AF_XDP Sockets for High Performance Network Function Chains. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3772052.3772258

1 Introduction

Fast network I/O techniques have significantly transformed the delivery of network services in recent years. Traditionally, packet processing applications such as switches, routers, firewalls, and load balancers were deployed as specialized hardware appliances, often referred to as hardware middleboxes. However, this model has been revolutionized by the emergence of software-based Network Functions (NFs), which run on commercially available off-the-shelf servers. Software NFs are easier to scale than hardware-based solutions and offer greater agility for development, maintenance, feature integration, and updates. Most importantly, they eliminate the problem of vendor lock-in, a significant concern with proprietary hardware-based solutions [2, 5, 7, 11, 12, 30, 31, 33].

Software NFs require fast network I/O to provide high throughput and low latency comparable to hardware middleboxes. Traditional kernel-based networking mechanisms, although reliable, introduce significant overheads, making them unsuitable for high-performance network environments, especially when bandwidths exceed tens or hundreds of gigabits per second [3]. High-speed network I/O frameworks like DPDK [10] address this issue by allowing userspace applications to interact directly with the Network Interface Card (NIC), bypassing the kernel stack completely to achieve near-line-rate packet processing. However, this approach also bypasses useful kernel features like packet filtering, queuing, and protocol processing. Further, DPDK operates in busy-polling mode to avoid interrupt overheads, resulting in high idle CPU usage. To mitigate this tradeoff, the Linux kernel community has introduced several features like the extended Berkeley Packet Filter (eBPF), eXpress Data Path (XDP), and AF_XDP [14, 17, 23]. Kernel-supported AF_XDP sockets enable low-overhead userspace access to raw packets through a partial kernel bypass mechanism — packets are first processed by the kernel's bottom-half interrupt handler, and then redirected to userspace (via an eBPF program in the driver's XDP hook) to skip further kernel processing. This design allows AF_XDP to achieve high packet processing rates, while retaining helpful kernel benefits (e.g., kernel control over NIC, low idle CPU usage due to interrupt-driven processing), making it a better choice than DPDK for usage in multi-tenant clouds [22]. In contrast, DPDK requires exclusive NIC ownership, necessitating either a centralized resource manager or virtualized NIC access, which introduces additional complexity and overhead in shared cloud infrastructures.

Network services are often instantiated as chains of multiple NFs co-located on the same host, to optimize both performance and resource utilization [33]. For example, a packet may traverse through a firewall, NAT, and a load balancer on a host. For high-performance packet processing, a packet I/O framework must support zero-copy packet transfers across trusted NFs co-located on the same host.

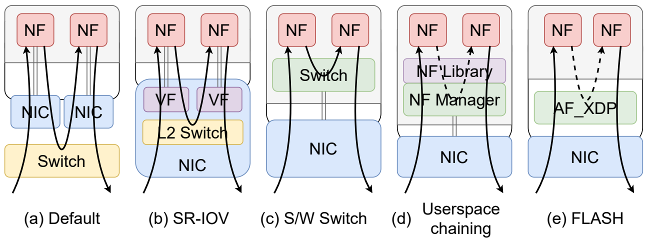

Figure 1: Kernel bypass packet delivery pipelines for co-located NFs. Solid lines denote packet copies; dotted lines indicate shared-memory zero-copy packet transfers.

However, no such option exists to efficiently chain AF_XDP applications today. By default, AF_XDP provides chaining via external switches (Figure 1a), SR-IOV-based Layer-2 switching inside the NIC (Figure 1b) [19], or software switches like Open vSwitch [40] (Figure 1c), all of which involve packet copies and hence degrade packet processing performance along the NF chain. DPDK-based frameworks like NetVM [19], OpenNetVM [45], and NetBricks [33] (Figure 1d) do provide userspace, zero-copy packet redirection, where an NF manager orchestrates the passing of packet descriptors along a chain of NFs written within the framework. However, extending these ideas to AF_XDP without modifying the userspace API is non-trivial because the packet datapath in AF_XDP is fragmented across the kernel driver and userspace application. For example, the various "rings" that transfer packet descriptors between the AF_XDP kernel driver and userspace application are implemented as shared lockless single-producer single-consumer rings. Therefore, for one NF to redirect shared-memory packet descriptors to another co-located NF's receive ring, the NF must synchronize with the kernel AF_XDP driver to access the ring, which is inefficient to implement in the existing AF_XDP datapath. To address this gap, this paper introduces FLASH, an in-kernel, zero-copy NF chaining framework for AF_XDP sockets (Figure 1e).

The FLASH framework consists of two components, one in the kernel and the other in userspace. FLASH extends the kernel AF_XDP datapath in several novel ways to enable seamless NF chaining, supporting zero-copy transfers between FLASH-native NFs and single-copy transfers for legacy AF_XDP NFs. FLASH extends the capabilities of the AF_XDP I/O rings to operate as multi-producer and multi-consumer lockless rings, while maintaining backward compatibility with the userspace AF_XDP APIs. This redesign relaxes the tight coupling between network queues and application endpoints, and enables multiple co-located application threads and the kernel driver to transfer packets using the rings. For correct operation during packet redirection, FLASH adds additional signaling mechanisms in the AF_XDP datapath, e.g., to wake an application thread after interrupt-processing by the kernel driver and when it receives packets from other co-located NFs. FLASH also incorporates several performance optimizations within the redesigned datapath, e.g., batching packet transfers to minimize synchronization overheads and backpressure detection to optimize CPU usage. In userspace, FLASH provides APIs and libraries (in C and Rust) that simplify the development of FLASH-native NFs and ensure packet isolation when used with shared-memory zero-copy packet transfers. FLASH-native NFs are built on unmodified AF_XDP APIs, and can also operate as standard AF_XDP applications without the FLASH related kernel modifications. FLASH uses a privileged process in userspace to separate the control plane operations in the applications (e.g., loading an XDP program) from the datapath, allowing FLASH-native NFs to run in unprivileged containers in cloud-based frameworks like Kubernetes. To the best of our knowledge, FLASH is the first NF chaining framework for AF_XDP sockets that provides optimized zero-copy packet transfers across co-located NFs natively within the AF_XDP datapath.

We port several popular common network functions to FLASH (e.g., a firewall [6], the Maglev load balancer [11], and the MICA key-value store [28]) and compare the performance of FLASH NF chaining with state-of-the-art chaining techniques. We find that FLASH matches the performance of popular DPDK-based userspace chaining frameworks, while retaining the benefits of the AF_XDP datapath that make it more suited than DPDK for multi-tenant cloud deployments. Further, FLASH outperforms existing techniques available for chaining AF_XDP applications (e.g., SR-IOV-based chaining), with 2.5× higher throughput and 77% lower latency.

The rest of the paper is organized as follows. We begin with background on AF_XDP and related work (§1), then outline the key challenges and design ideas behind FLASH (§2). Next, we present FLASH's architecture (§3) and implementation (§4), followed by a detailed evaluation using microbenchmarks and real-world NF workloads (§5). We conclude with key takeaways and directions for future work (§6). FLASH is open-source and is available at https://github.com/networkedsystemsIITB/flash.

2 Background and Related Work

This section introduces AF_XDP sockets as a promising alternative for high-performance NF deployments and reviews related work on NF chaining approaches.

2.1 AF_XDP Overview

AF_XDP is a Linux socket family designed to facilitate high-performance packet processing in userspace, while addressing many of the limitations of complete kernel-bypass frameworks like DPDK. Ethernet NICs interact with the kernel via a NIC driver, which registers a function (`napi_poll`) for packet reception and transmission. This function is executed by the kernel's bottom-half interrupt handler, the `ksoftirqd` thread, which is scheduled dynamically based on network load. Network packets are first handled by `ksoftirqd`, and then passed over to the kernel network stack for further processing. AF_XDP optimizes this process using an eBPF hook, known as eXpress Data Path (XDP), which runs within the `napi_poll` function of the kernel driver. XDP can be used to intercept incoming packets and forward them to an AF_XDP socket, bypassing the kernel stack processing. Outgoing packets are delivered directly to `napi_poll` for transmission. This technique provides the benefits of kernel-bypass frameworks while ensuring that the kernel retains some level of control over the packet datapath, e.g., enabling only specific packet flows to bypass the kernel, while directing others through the standard kernel network stack.

Figure 2 shows the packet processing flow of a packet forwarding application using AF_XDP. At initialization, the AF_XDP userspace application allocates a contiguous memory region, called UMEM,

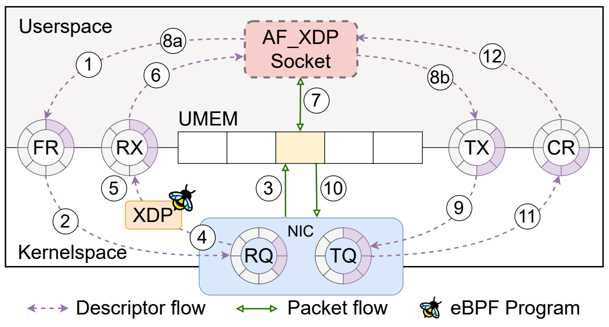

Figure 2: AF_XDP packet processing pipeline.

to buffer network packets. This memory is shared with the driver in the kernel, enabling the NIC driver to perform DMA (Direct Memory Access) operations directly into the UMEM, avoiding costly packet copies between kernel and userspace. Four single-producer single-consumer (SPSC) circular rings, namely Fill ring (FR), Completion ring (CR), Rx ring (RX), and Tx ring (TX), allocated by the kernel and shared with the userspace application, facilitate the coordination of buffer accesses between the userspace application and the driver. To receive packets into the UMEM, the userspace application populates the FR with descriptors (pointers) to empty buffers in the UMEM (1), signaling the NIC driver they're ready for use. The NIC driver dequeues these descriptors and enqueues them into the NIC's Receive queue (RQ) (2). Upon receiving a packet, the NIC performs a DMA transfer directly into the buffer referenced by a descriptor in the RQ (3). An eBPF program is executed in the driver at the XDP hook for each packet, determining whether the packets should be redirected to an AF_XDP socket or the kernel network stack (4). The driver then places a descriptor to the packet buffer in the socket's RX ring (5). The userspace application later retrieves the packet from the RX ring (6) and processes it (7). If the packet is not required to be transmitted, the descriptor to the packet buffer is returned to the FR (8a). If the packet is to be transmitted, the application places the buffer's descriptor into the TX ring (8b), the NIC driver then dequeues packets from the TX ring and enqueues them for transmission in the NIC's Tx queue (TQ) (9). The DMA engine then transmits the packet from the TQ of the NIC (10). Once the packet is transmitted, the descriptor is returned to the CR by the driver (11), which the application uses to replenish the FR for receiving packets in the future (12). This technique of redirecting packets from kernel to userspace and vice versa using a reference allows zero-copy packet handling in AF_XDP.

Each AF_XDP socket is bound to a specific combination of a network device (netdev) and a hardware queue, identified as a {netdev, queue} pair. Each socket typically uses an RX and a TX ring, while one FR and one CR are required per {netdev, queue} pair. This ensures that each queue, processed by separate per-CPU `ksoftirqd` instances, maintains SPSC ring semantics.

An application can interact with the AF_XDP driver in several *modes*. In the *default mode* of AF_XDP, the kernel driver code invoked by the `ksoftirqd` thread populates the RX ring of the socket, and the userspace application thread monitors this ring in a busy loop. In *poll mode*, userspace applications use system calls like `poll()` to block for new packet arrivals in the RX ring, avoiding busy waiting inside the application. Both the default mode and poll mode described above rely on interrupt-driven processing by the `ksoftirqd` thread in the kernel. There is also a third *busy-poll mode* in AF_XDP, where interrupts are disabled, and the application polls packets directly from the NIC, much like DPDK. In this mode, a separate `ksoftirqd` thread is not required, and the `napi_poll` function is executed in the kernel when the user application makes system calls to send and receive packets. While the interrupt-driven modes benefit from a low idle CPU usage, the busy-poll mode is more efficient when the incoming packet rate is high, as it avoids context switches between user and kernel threads. Therefore, an AF_XDP application can adopt a hybrid design, where it uses busy polling under high load, and the blocking `poll` system call when under low load [22], much like the NAPI-based design of the `ksoftirqd` thread in the Linux kernel.

2.2 Related Work

DPDK-based frameworks for NF chaining. Since in-kernel network stack processing overheads are unsuitable for NFs, several kernel bypass frameworks have been developed to enable direct packet processing in userspace. DPDK-based frameworks such as NetVM [19], OpenNetVM [45], NetBricks [33], and MiddleNet [44] support NF chaining in zero-copy mode by passing packet references between functions. These frameworks require NFs to be written using specific libraries and APIs, which manage packet flow across the NFs. While many of these frameworks offer memory isolation through containers or virtual machines, they often lack packet isolation, allowing NFs to access or modify packets after forwarding. An exception is NetBricks, which ensures both memory and packet isolation by executing all NFs within a Rust-based safe runtime. Open vSwitch (OVS) [40], which supports DPDK as a backend, enables NF chaining but relies on single-copy packet forwarding, leading to suboptimal performance.

Despite performance benefits like zero-copy packet transfers, DPDK-based NF chaining frameworks have significant drawbacks. They rely on busy polling in userspace and require exclusive NIC control, completely bypassing the kernel's networking stack. This results in high idle CPU utilization and prevents the NIC from being shared with other kernel-based applications or legacy NFs. Deploying DPDK-based userspace chaining solutions across separate NF processes requires turning off security features like Address Space Layout Randomization (ASLR) [8], posing serious security risks in multi-tenant environments. In contrast, FLASH provides kernel-native zero-copy support, along with low idle CPU usage and compatibility with legacy AF_XDP NFs (via a single-copy mode).

AF_XDP based NF chaining. OVS also supports AF_XDP sockets as a backend, which suffers from inefficiencies similar to its DPDK counterpart, due to single-copy packet forwarding. Previous work [39] explored AF_XDP over SR-IOV (AF_XDP-SRIOV) for telecom workloads, combining flexibility and memory isolation, without losing kernel control. SR-IOV-based chaining introduces challenges such as IOTLB misses in long NF chains [1, 19], and packet copies between NFs, which increase latency and reduce throughput. xsk-nf [34] proposed an alternative design that chains AF_XDP-based NFs within a single process using a run-to-completion model. But xsk-nf's monolithic architecture supports

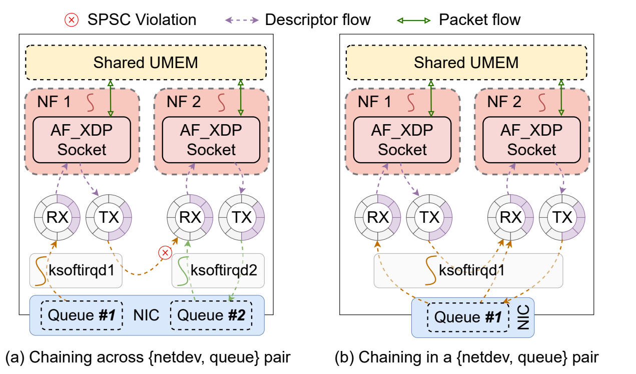

Figure 3: (a) Example of SPSC rings semantic violation when chaining logic is introduced in multi-queue AF_XDP setups. (b) An example of SPSC rings supported chaining with multiple sockets sharing a single queue.

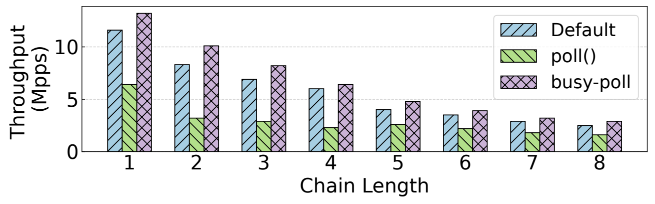

Figure 4: Throughput of SPSC-supported chains with varying chain lengths with 64B packet size.

only static chains, limiting scalability and flexibility in dynamic, cloud-native deployments. There is no AF_XDP solution that provides flexible yet performant NF chaining like FLASH.

Optimization of NF chains. Several research efforts have explored optimizations to mitigate performance bottlenecks in NF chaining. NFVnice [25] introduced a scheduling and backpressure mechanism that dynamically adjusts NF scheduling priorities based on system load and network conditions, thereby improving CPU utilization and throughput. Complementary approaches include REINFORCE [24], which enhances resilience through NF chain replication; DHL [27], which leverages FPGAs to accelerate NFV deployments; and FlexNFV [13], which extends OpenNetVM to support elastic scaling. Other efforts, such as Reuse [20], NFVDeep [43], which optimizes NF chain deployment strategies, and LemonNFV [26], which enables heterogeneous NFs to coexist on the same host, further advance efficiency and scalability. Collectively, these strategies help reduce resource contention, improve packet processing performance, and enhance scalability. While the above frameworks are designed for DPDK their techniques do not readily extend to AF_XDP.

Other zero-copy frameworks. A separate class of zero-copy frameworks targets serverless deployments over the transport layer. Spright [36], for example, uses a gateway where packets are processed by the kernel stack and their payloads are placed into shared memory. Inter-function communication is then handled via busy-polling or eBPF. However, this design is not truly zero-copy, as data is still copied at the gateway, and eBPF is used only to coordinate inter-function events after the payload has been placed in shared memory. In contrast, our system extends AF_XDP to provide end-to-end zero-copy across applications. Other frameworks like SURE [35] use DPDK for transport-layer zero-copy, inheriting DPDK's drawbacks, while Palladium [37] leverages RDMA for multi-node setups. None of these supports general-purpose network functions (NFs) at Layers 2–4. Our framework enables zero-copy processing across Layers 2–4 and is easily extensible to higher layers. Separately, there are frameworks like Polycube [32], which chains privileged eBPF-based NFs via eBPF tail calls—an approach unsuitable for multi-tenant environments. Instead, we support chaining of unprivileged userspace programs that can be isolated amongst themselves, making our design safer and more practical for the cloud.

3 Design Challenges and Key Ideas

AF_XDP sockets can reuse a socket descriptor when binding to a socket, to share a UMEM region [21]. While this feature allows for reusing the UMEM across sockets, it also opens up the possibility of zero-copy chaining between NFs, where a packet transmitted by one NF can be directly received by another without memory copies. We refer to this mechanism as ***zero-copy packet redirection***. However, the current Linux kernel lacks any mechanism—zero-copy or otherwise—for in-kernel redirection between AF_XDP sockets, and once a packet is transmitted by a socket, it is immediately sent to the NIC. This limitation prevents efficient chaining of AF_XDP-based NFs. FLASH addresses this gap by introducing in-kernel support for chaining AF_XDP sockets. Below, we discuss the challenges in realizing our goal and the key ideas we bring to the table.

Challenge 1: Lockless SPSC rings of AF_XDP. AF_XDP relies on lockless Single-Producer Single-Consumer (SPSC) rings to achieve high performance [18]. A straightforward mechanism to redirect packets across AF_XDP socket {netdev, queue} pairs violates the SPSC semantics of the rings. Consider the example in Figure 3a, where NF1 transmits a packet to NF2. Now, updates to the NF2's RX ring are subject to a race condition between the actions of `ksoftirqd1` and `ksoftirqd2`, and the SPSC ring will not work. An alternative approach is to bind multiple sockets to the same {netdev, queue} pair (as shown in Figure 3b). This design preserves the default SPSC semantics of the rings, since a single `ksoftirqd` thread handles and serializes all kernel operations. However, this approach suffers from severe performance degradation as the NF chain length increases, across all AF_XDP modes, as shown in Figure 4. The bottleneck arises because the single `ksoftirqd` thread processes all packets coming to the different NFs.

To overcome the problem described above, FLASH modifies the semantics of access to AF_XDP rings to support multiple producers or multiple consumers, but without affecting the existing userspace API, thereby ensuring backward compatibility with current applications. Now, when the rings of a single NF are accessed and updated by `ksoftirqd` threads of multiple NFs executing on different cores, an update of a shared pointer of the head/tail of the MP/MC ring leads to cache invalidation on all other cores and a drop in performance. Therefore, FLASH performs redirection in batches of packets to amortize the overhead incurred with MP/MC rings.

Challenge 2: Fragmented AF_XDP datapath. AF_XDP is not a complete kernel bypass mechanism like DPDK by design, and as a result, its packet datapath is fragmented across the kernel and userspace. This fragmentation has several implications for efficient packet chaining. For example, when FLASH moves consumed packet descriptors from a downstream NF to an upstream NF (to maintain the balance of free descriptors along the chain), the operation is complicated by the fact that the free packet buffer pool is split across the completion ring (CR) populated by the kernel, and the fill ring (FR) populated by userspace, unlike in DPDK that manages the free buffer pool fully in userspace.

FLASH comes up with several novel ideas to work around these constraints of the AF_XDP datapath design. For example, when redirecting batches of packets across NFs, FLASH updates the rings in a very specific order to avoid race conditions between user and kernel threads. FLASH also uses additional temporary pools of free buffers between NFs to simplify the synchronization.

Challenge 3: Scheduling of application and kernel threads. NF chaining faces several challenges with how the AF_XDP application and `ksoftirqd` threads are scheduled in the AF_XDP datapath. If network packets are not received or transmitted to/from an NF using the NIC, and no interrupts are triggered, then the `ksoftirqd` thread servicing that socket is not scheduled by the OS, and the `napi_poll` function does not execute. However, in the case of NF chaining, the `ksoftirqd` thread is also responsible for handling packet redirection and must be scheduled even in the absence of NIC interrupts. But with the current AF_XDP datapath, for longer NF chains where intermediate NFs do not directly interact with the NIC, the `ksoftirqd` threads of NFs downstream in the chain are not scheduled correctly, leading to failure in packet forwarding. Conversely, if an NF is redirecting packets to a slower downstream NF that is not able to process packets fast enough, the upstream NF threads need not be scheduled as much, but the current AF_XDP datapath has no such mechanisms to apply backpressure and throttle upstream NFs, resulting in wasted CPU cycles.

To overcome this challenge, FLASH makes changes to the way the AF_XDP application and `ksoftirqd` threads are scheduled in the current AF_XDP datapath, to propagate packet redirection signals down, and backpressure signals up the NF chain correctly. These changes ensure that application and driver threads get scheduled enough to guarantee good throughput while avoiding wasting CPU cycles due to slower downstream NFs.

Challenge 4: Zero-copy performance vs. multi-tenant safety. AF_XDP NFs require elevated privilege for multiple tasks, e.g., creating AF_XDP sockets, loading the XDP program in the kernel. However, executing NFs at a higher privilege level also exposes the NFs in the chain to malicious or benign buggy behavior from other NFs. Moreover, zero-copy packet redirection requires unfettered access to packet data in the shared UMEM, which may be tricky in a multi-tenant cloud deployment where tenants may not trust the NFs from each other, but would still like to reap the performance benefits of zero-copy packet transfers via shared memory.

FLASH mitigates this tradeoff between performance and security in two ways. First, FLASH separates out the control path operations of an NF (e.g., creation of AF_XDP sockets) from the datapath (actual packet processing), and a separate privileged FLASH “monitor” process handles all privileged operations on behalf of the FLASH-native NFs, which allows the NFs themselves to run in unprivileged containers or baremetal processes. Second, FLASH provides APIs and libraries to NF developers, which abstract away all the control path setup and communication with the FLASH monitor, allowing developers to focus only on the NF packet processing logic. We provide APIs in both C and Rust, with the Rust library ensuring memory isolation at the packet granularity using the language's compile-time memory safety features, while allowing zero-copy packet redirections. Of course, because FLASH does not change the AF_XDP userspace APIs in any way, and all redirection logic is implemented only in the kernel, legacy NFs not built with the FLASH library (and not sharing UMEM with FLASH-native NFs) can still communicate with FLASH-native NFs via single-copy packet redirection provided by FLASH in the kernel.

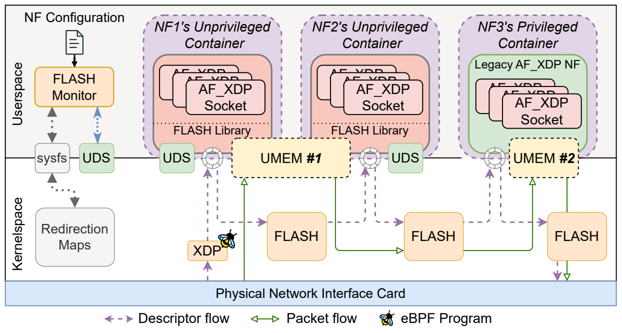

Figure 5: Overall architecture of FLASH. NF1 and NF2 use zero-copy chaining, and NF3 uses single-copy chaining.

4 Design of FLASH

This section describes the design of FLASH within the Linux kernel's AF_XDP subsystem and its userspace components. We begin with an overview of the proposed solution.

4.1 System Architecture

The overall architecture of FLASH is illustrated in Figure 5. FLASH provides a userspace daemon, known as the FLASH monitor, to handle all privileged control-plane operations, including creating AF_XDP sockets, loading XDP programs, and setting redirection paths within the kernel. The NF chain topology (e.g., how many NFs exist, which NF can potentially communicate with which other NF, and whether to use zero-copy or single-copy mode) and runtime parameters (e.g., the netdev, queue pair each NF should bind to, and the number of threads it can use) are provided to the monitor via a configuration file. The monitor uses this information to set up redirection paths in the kernel via the `sysfs` interface (§5.1). The NF chains enabled by FLASH are dynamic and flexible—when the NF topology allows an NF to redirect packets to more than one downstream NF, the upstream NF places metadata in the packet descriptor that helps the kernel redirection logic choose the destination downstream NF. NFs communicate with the monitor through a Unix Domain Socket (UDS) [29] to exchange shared memory file descriptors and other metadata, using which the NFs obtain access to the shared UMEM. With the monitor handling all privileged tasks, the framework supports both bare-metal NF chains and unprivileged container-based deployments for multi-tenant clouds.

To simplify NF development and integration, FLASH exposes APIs and libraries in both C and Rust. Additional libraries for other languages can be developed with minimal effort. For deployments that require strong packet-level isolation with zero-copy support, where an NF must only access packets it has received, FLASH offers a Rust library that exposes safe APIs for NFs to allocate, receive, transmit, or drop packets. The NF developed with the FLASH Rust library is granted access to a fixed-size, zero-copy mutable array for each received packet, accessible only until the packet is transmitted or dropped. NFs implemented in Rust also benefit from the language's ownership model, compile-time enforcement of thread safety, and robust error handling that prevents common runtime failures. The C language APIs of FLASH do not provide similar isolation guarantees, but have better performance. Isolation can also be achieved using single-copy chaining in any language, though this comes with reduced performance due to additional packet copies.

Because FLASH provides unmodified AF_XDP APIs in the datapath, legacy applications are supported without modification, and can communicate with FLASH-native NFs in single-copy mode. These applications do not need to interact with the monitor directly, as in-kernel redirection is managed transparently in the kernel. However, these NFs may need to run at a higher privilege level to load the XDP program and other tasks.

Within the kernel's AF_XDP subsystem, FLASH intelligently re-architects the ring buffers and the packet flow through them to enable packet redirection in either zero-copy or single-copy mode (§4.2). To enhance throughput, packets are batched during redirection, and partial redirection of batches is handled carefully when downstream NFs do not have enough space in their rings (§4.3). To further improve efficiency, FLASH changes the scheduling logic of the AF_XDP application and `ksoftirqd` threads to enable efficient sleep and wakeup of the threads, in the face of variable incoming packet load and processing speeds of the NFs along the chain. These performance optimizations are implemented across the kernel and userspace FLASH library, allowing NFs to leverage performance gains without changes to the NF logic itself (§4.4).

4.2 Packet Redirection Inside the Kernel

FLASH implements zero-copy and single-copy packet transfer logic between NFs within driver-agnostic AF_XDP kernel code. This code is invoked from the driver's `napi_poll` function, which is executed by the `ksoftirqd` thread during packet transmission and reception. Figure 6 shows the zero-copy packet flow from NF1 to NF2 that share a UMEM, and single-copy transfer between NF2 and NF3 that do not share a UMEM.

Zero-copy packet redirection. We will first describe the steps involved in zero-copy packet transfer between NFs. Steps (1) to (8) align with the packet flow shown in Figure 2, where packets are received based on the {netdev, queue} pair to which the socket is bound. Hardware filters such as n-tuple filters can be configured in the NIC for this setup [16]. However, instead of immediately transmitting the packet through the NIC, FLASH determines whether redirection is needed (9). If redirection is not required, the packet is sent out via the NIC, as shown for NF3 in the figure. However, if redirection is necessary, the descriptor must be enqueued into the RX of the next socket in the NF chain. This enqueuing operation disrupts the balance of packet descriptors in the respective pools—NF1 has one less descriptor, and NF2 has a descriptor in the RX without consuming a buffer from its pool. To restore balance, before

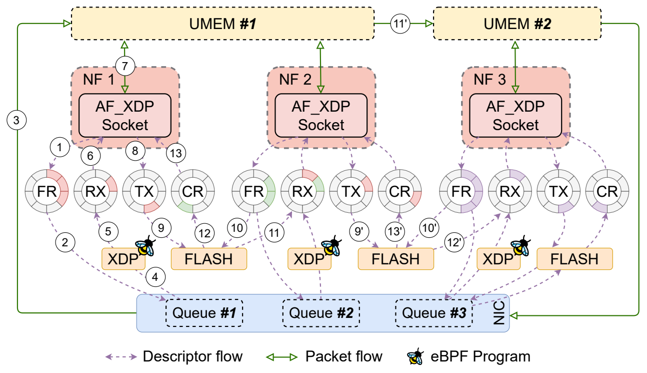

Figure 6: Packet redirection flow with FLASH. NF1 and NF2 share a common UMEM, and NF3 uses a separate UMEM. NF1 to NF2 is zero-copy, and NF2 to NF3 is single-copy.

enqueuing the descriptor to NF2's RX, a new descriptor is retrieved from NF2's FR (10). Once this new descriptor is available, the packet is enqueued into NF2's RX (11). The original descriptor, which was meant for transmission, is then swapped with the new descriptor and enqueued into NF1's CR (12), after which NF1's application returns the descriptor to its FR (13). This completes the redirection between NF1 and NF2.

From the perspective of the first NF, the packet transmission process appears unchanged—a packet is passed using the AF_XDP socket for transmission to the NIC. In reality, the packet is seamlessly handed off to the next NF, where the packet is treated as a newly received packet. From the figure, it can be observed that there are multiple consumers of the FRs and multiple producers of RX rings. Because these rings are updated asynchronously by separate `ksoftirqd` threads, the single-producer single-consumer (SPSC) rings of AF_XDP do not suffice. The FR must be updated to single-producer multi-consumer (SPMC), and the RX must be updated to multi-producer single-consumer (MPSC). However, since all rings are single-producer or single-consumer in userspace, FLASH does not change the userspace AF_XDP API and remains backward compatible with legacy applications.

Single-copy packet redirection. The single-copy packet redirection of FLASH is designed for scenarios where NFs do not share the UMEM region. Unlike DPDK-based userspace chaining solutions that require NFs to be implemented using custom libraries, this approach enables the chaining of legacy AF_XDP NFs, which inherently do not share UMEM with other NFs.

The packet flow for single-copy redirection is also depicted in Figure 6 between NF2 and NF3. Steps (1) to (8) remain identical to those outlined in Figure 2. However, after determining that a packet requires redirection (9'), a descriptor is consumed from the FR of the target socket (NF3) (10'). The packet is then copied to the new memory address from the descriptor using `memcpy()` (11') and this new descriptor is added to the target socket's RX (12'). The original descriptor is returned to the original socket (NF2), which subsequently recycles the descriptor back into its own FR, completing the cycle (13'). As in the zero-copy design, the FRs in this scheme must operate in a single-producer multi-consumer (SPMC) mode, while the RX rings must operate in a multi-producer single-consumer (MPSC) mode.

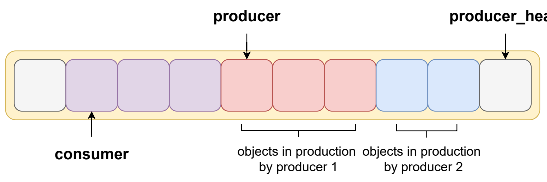
producer
producer_head
consumer
objects in production by producer 1
objects in production by producer 2

Figure 7: A state where two producers attempted to claim space at the same time, with CAS of producer 1 succeeding first and producer 2 retrying to get the next entries.

Extending SPSC Rings to MPSC and SPMC. The existing AF_XDP rings use a lockless single-producer single-consumer (SPSC) model, where a single producer writes data into a ring buffer, and a single consumer reads from the ring buffer [4, 18]. The ring operates as a circular buffer with a pre-allocated power-of-two-sized array. Two global pointers, *producer* and *consumer*, track where data is added and consumed. To reduce costly global accesses, each side maintains local copies of these pointers for efficient operation.

We extend the default design to a multi-producer single-consumer (MPSC) ring for the RX as follows [9, 15]. A new global pointer, *producer_head*, is introduced to serialize concurrent production. When a producer adds data, the producer reserves space using an atomic Compare and Swap (CAS) operation, ensuring that only one producer succeeds per attempt while others retry (illustrated in Figure 7). Once data is written, the producer updates the *producer* pointer, maintaining correct order across multiple producers. The consumer, which operates from userspace, remains unchanged, requiring access to only the *producer* pointer without the knowledge of the *producer_head* pointer. Similarly, we implement a single-producer multi-consumer (SPMC) ring for the FR by introducing a new *consumer_head* pointer to serialize consumer accesses while the userspace producer operation remains unchanged.

4.3 Batched Packet Redirection

The above redirection mechanism can lead to non-local ring pointer accesses when NFs are pinned to run on separate cores. These cross-CPU accesses cause cache invalidations and degrade performance, especially when redirection is performed per packet. To mitigate this, FLASH defers redirection until the `napi_poll` completes processing a batch of packets. Descriptors of packets to be redirected are copied to a pre-allocated array during transmission (shown as temporary TX buffer in Figure 8). After `napi_poll` completes, all redirections are handled in a single batch, reducing ring accesses and consolidating pointer updates into fewer atomic operations.

While batching redirections improves performance, it also introduces challenges in buffer management with AF_XDP. Unlike DPDK, which handles buffer pools entirely in userspace, AF_XDP splits buffer management between the kernel and userspace. This fragmented design means that redirection operations using multi-producer single-consumer (MPSC) and single-producer multi-consumer (SPMC) interactions, especially with batched packets, must be handled carefully when one or more rings involved do not have enough available space or descriptors. For example, if a network function (NF) wants to send five packets, but the receiving NF's RX ring only has room for two, only two packets can be redirected. More generally, when an NF attempts to transmit n_{tx} packets, the

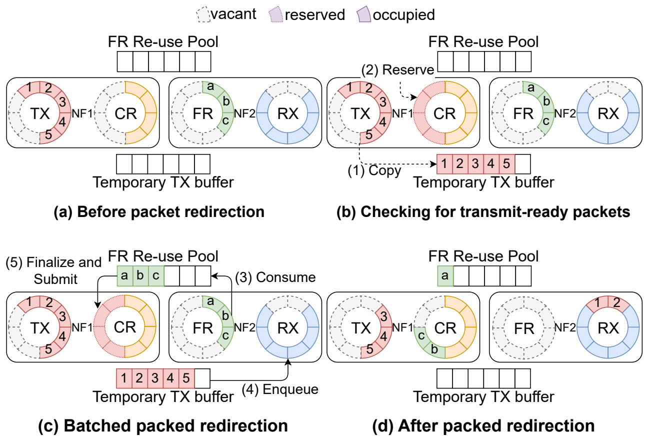

Figure 8: Batching of packet redirection when $n_{tx} = 5$, $n_{cr} = 4$, $n_{fr} = 3$, and $n_{rx} = 2$.

actual number of packets that can be redirected depends on three factors: vacant entries in its CR (n_{cr}), available descriptors in the next NF's FR (n_{fr}), and vacant entries in next NF's RX (n_{rx}). The minimum of these values determines how many packets can be successfully redirected. For instance, if $n_{tx} = 5$, but $n_{cr} = 4$, $n_{fr} = 3$, and $n_{rx} = 2$, then only two packets can be redirected. The redirection status of all packets in a batch has to be preserved across subsequent `napi_poll` invocations. Note that the current `napi_poll` design does not support partial transmission of a batch of packets.

Further, because FR and RX rings are concurrently accessed by multiple NFs, their available capacities may change during redirection. As a result, the exact number of packets that can be redirected cannot be determined upfront. One approach is to optimistically begin with n_{tx} packets and adjust dynamically if subsequent rings have limited availability. This strategy, however, requires handling overcommitment from earlier stages. For example, if you enqueue more packets into the RX ring of the downstream NF than there are free descriptors in its FR, then the redirection cannot proceed correctly. With SPSC rings, like TX and CR, overcommitment is easily reversible since descriptors are only peeked or reserved locally before being globally committed. In contrast, FR and RX are SPMC/MPSC rings where descriptor peeks (FR) and reservations (RX) require a global commit, making it harder to roll back without violating ring semantics. To address this challenge, FLASH uses two ideas—first, the rings are accessed and updated in a specific order that makes it easier to handle overcommitments, and second, FLASH uses a temporary buffer to store any surplus descriptors and reuses them in future iterations.

The batched packet redirection technique used by FLASH is illustrated with an example in Figure 8. When the driver checks for transmit-ready packets (specifically using `xsk_tx_peek_desc` function), FLASH extends `napi_poll` to do the following: (1) copy descriptors to the temporary TX buffer and (2) reserve space in the CR. If the CR does not have enough space, only the available slots are used (e.g., 4 out of 5 descriptors), and the number of packets for redirection is adjusted. Later, when there are packets for transmission or redirection (in `xdp_do_flush`), FLASH does the following: (3) consumes descriptors from the next NF's FR, (4) enqueues packets into the next NF's RX, and (5) finalizes and submits

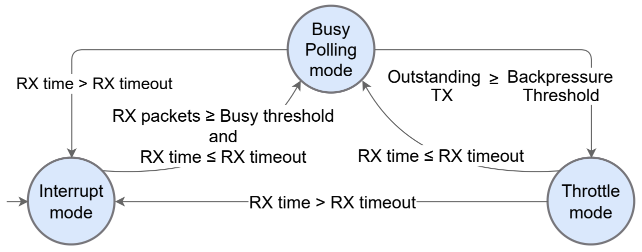

Figure 9: Early backpressure detection based smart polling.

CR reservations. This ordering ensures packets are redirected only if space and buffers are available throughout the pipeline. If steps (3) or (4) fail for some packets in the batch, any earlier CR reservations from step (2) are rolled back. Because descriptors consumed from the FR cannot be rolled back easily, leftover FR descriptors are saved temporarily and reused in the next iteration. Only successfully redirected packets are cleared from the TX ring; others remain and are retried later.

4.4 Scheduling User and Kernel Threads

Inspired by earlier frameworks that use AF_XDP in cloud deployments [22], and by how `napi_poll` is scheduled in the Linux kernel itself, the FLASH library uses a smart polling mechanism, where the application dynamically switches between the busy polling mode of AF_XDP, and the interrupt-driven `poll()` mode. When no packets are available, the FLASH userspace library invokes the `poll()` system call, putting the NF into a blocked state and transitioning into interrupt mode. When the NIC receives a packet, the kernel handles the interrupt and marks the corresponding socket as readable. This causes `poll()` to return and wake up the NF, allowing it to resume busy polling and process incoming packets. As illustrated in Figure 9, each NF starts in interrupt mode, switches to busy polling when packet arrival exceeds a threshold, and reverts to interrupt mode after a timeout. Both the threshold and timeout are fully configurable, allowing NFs to adapt dynamically based on workload characteristics and performance goals.

However, this dynamic switching between interrupt and busy polling modes of AF_XDP does not work well to handle NF chaining. For downstream NFs in a chain, the interrupt-driven `ksoftirqd` thread or the application thread blocked in `poll()` only wake up when a packet arrives from the NIC, and not when packets are redirected from an upstream NF. Therefore, FLASH changes the thread scheduling in the AF_XDP datapath, to propagate the wakeup signal down the NF chain even on packet redirections. The FLASH kernel marks the destination NF's socket as readable after redirecting a packet, causing the next NF's `poll()` to return.

Further, as discussed in §4.3, if packet redirection fails due to the next NF's rings being full or empty, our kernel implementation ensures that the packet remains in the TX ring of the current NF. This behavior naturally creates backpressure upstream, preventing the source NF from sending additional packets. However, continuous transmission retries in the throttled NF result in excessive CPU usage. To address this inefficiency, FLASH introduces an early backpressure detection mechanism in the userspace library. As shown in Figure 9, when backpressure is detected, the NF does not spin indefinitely. Instead, it enters a Throttle mode, where it temporarily pauses transmission—typically by sleeping for a fixed duration, before rechecking whether the congestion has cleared. If backpressure persists, the NF sleeps again; otherwise, it resumes packet transmission. The backpressure detection threshold is configurable per NF, enabling developers to tune behavior based on workload characteristics and performance goals. Overall, this strategy reduces unnecessary CPU usage during congestion while maintaining smooth packet flow across the NF chain. As part of ongoing work, we are enhancing this mechanism to support blocking with `poll()` instead of sleeping, and the downstream kernel notifies the upstream NF when congestion clears by marking the socket writable, enabling efficient, event-driven transmission.

5 Implementation

The complete FLASH implementation consists of 16,000 lines of C and Rust code, spanning both userspace and kernelspace components. Of these, 700 LoC represent modifications to the Linux kernel. We have integrated FLASH with most AF_XDP-supported drivers, including widely-used network drivers like Intel's `ixgbe`, `i40e`, and `ice`, as well as NVIDIA / Mellanox's `mlx5` driver.

5.1 Packet redirection via sysfs

FLASH introduces a new sysfs interface that allows the FLASH monitor to identify AF_XDP sockets and configure redirection rules. To support flexible and scalable redirection, FLASH lets the userspace monitor associate every socket with one or more next-hop destination sockets. Since AF_XDP sockets are referenced using process-local file descriptors, FLASH introduces a process-independent identifier called **flash-id**. Each socket is assigned a unique flash-id, using Linux's ID allocator (idr) [41], which internally maps to the socket's kernel pointer—enabling consistent, cross-process identification and redirection of packets within the kernel. FLASH exposes these flash-ids and their associated metadata—such as netdev, queue, process ID, and process name—via the `sysfs` interface. When a new AF_XDP socket is created, FLASH automatically creates a corresponding `sysfs` directory. Each directory contains a writable file named *next*, which is used to configure redirection targets. By default, this file contains -1, indicating that the socket does not redirect packets but transmits them directly to the NIC using the default AF_XDP data path. Privileged users can write one or more flash-ids to this file to enable redirection. Internally, FLASH maintains a redirection map—an array of socket pointers—for each AF_XDP socket. When flash-ids are written to the *next* file, FLASH resolves them to the corresponding socket pointers and updates this map. During packet transmission, the kernel looks up the redirection map to determine where to forward the packet. The packet is redirected there by default if only a single destination is present. For complex chains involving multiple downstream sockets (e.g., branching chains), userspace applications must specify the redirection target by writing the index of the destination in the redirection map into the first 16 bits of the packet descriptor's flags field, which is read by FLASH to perform redirections. To simplify deployment, FLASH removes the need for users to manually configure redirection rules through `sysfs`. Instead, all configurations are automatically managed by the FLASH monitor based on a user-provided configuration file. Legacy NFs that do not integrate with the FLASH can manually write redirection rules to the appropriate *next* files to connect legacy NFs into existing chains. This capability ensures backward compatibility and eases transitioning from traditional AF_XDP applications to FLASH-enabled setups.

5.2 FLASH userspace library and API

NFs interact with AF_XDP sockets using standard system calls including `socket()`, `bind()`, `getsockopt()`, `setsockopt()`, `mmap()`, `poll()`, `sendto()` and `recvfrom()`. However, these system calls are primarily used for setting up resources—such as socket creation, ring configuration, memory mapping, and polling—rather than for actual packet transmission or reception. The responsibility for managing descriptors in userspace, including sending and receiving packets, is left to the application itself. While this approach enables fine-grained control over packet processing, it is often complex and prone to errors. To make NF development easier, FLASH provides userspace libraries in both C and Rust that hide the complexity of system calls and the data path. These libraries communicate with the FLASH monitor, set up rings, and map memory. They also offer a set of datapath APIs for receiving, sending, dropping, and allocating descriptors. With these APIs, developers can build sophisticated NFs or even full transport protocols. Both libraries are built for performance, using thread-local data structures to eliminate locks and enable easy NF scalability. Additionally, the Rust library enforces strong memory safety and isolation through the language's ownership model. NFs access packet data only via the FLASH API, ensuring that a mutable reference to a packet can be held by only one NF at a time. Packet ownership is transferred by the kernel during redirection, preventing concurrent access and safe zero-copy operation without sacrificing performance.

5.3 FLASH monitor

When an NF connects to the FLASH monitor, the monitor allocates a dedicated and isolated UMEM region for that NF. This allocation is based on parameters specified in a configuration file provided to the monitor. In cases where multiple NFs are intended to share a memory region and operate in zero-copy mode, the UMEM is shared; however, each NF is restricted to accessing only its designated portion of the shared region. This ensures robust memory isolation, allowing each NF to function independently without interference. For copy-based setups, rather than sharing a UMEM region, each NF is assigned a separate UMEM, further reinforcing isolation. Additionally, the FLASH monitor supports both single and multi-NIC configurations and can optionally enable packet redirection to the host network stack when required.

FLASH supports containerized deployments using Docker and Kubernetes. In this architecture, the Flash monitor runs inside a privileged container and handles privileged operations, while NFs are executed within unprivileged containers. This setup ensures strong isolation, high performance, and operational flexibility when deploying and managing network functions. With Kubernetes, the monitor runs as a device plugin [38] on each node, allowing the entire NF topology to be configured centrally—even across multi-node clusters. When deployed as a K8s DaemonSet, the device plugin reads a node-specific configuration file to initialize the monitor accordingly. NIC queues are exposed as Kubernetes resources, providing fine-grained control over their allocation and enabling efficient sharing in multi-tenant environments. Once the setup is in place, users can deploy NFs as standard Kubernetes pods without needing to manage low-level network or device configurations. When defining a pod, the AF_XDP resource must be specified in the pod specification. If permitted by the configuration, the pod can then run a FLASH NF seamlessly, without any modifications to the NF. Importantly, this architecture allows the NIC to be used simultaneously for other networking purposes beyond the NF deployment. This level of flexibility is a key advantage over DPDK-based setups, which often require exclusive NIC access.

5.4 Network Functions with FLASH

To show the usability of our framework, we have implemented the following well-known NFs from the literature: (i) **Firewall.** A firewall that performs a sequential lookup on ACL to find rules to drop packets [6]. (ii) **Maglev.** A widely-used load balancer developed by Google, Maglev [11] efficiently distributes traffic across multiple backends using consistent hashing. (iii) **MICA.** A zero-copy, in-memory key-value store that leverages kernel bypass, parallel request handling, and optimized data structures to deliver high throughput for read-write intensive workloads [28]. (iv) A simple **L2fwd** NF that redirects packets based on the MAC address. (v) An **ARP Responder** that handles ARP requests. (vi) A **Ping Responder** that handles ICMP echo requests.

6 Evaluation

In this section, we evaluate FLASH's performance and efficiency compared to other NF chaining solutions. We also assess the impact of individual FLASH features and quantify their overheads.

FLASH variants. The following FLASH variants were used for evaluation: **FLASH-ZC**, zero-copy chaining using our C library, without any packet isolation. **FLASH-ZR**, zero-copy chaining with packet isolation using the Rust library. **FLASH-SC**, single-copy chaining with packet isolation via the C library.

Baselines. AF_XDP-US represents an upper-bound ideal implementation where multiple AF_XDP-based NFs run as threads within a single process. Packets are received through sockets and forwarded between threads using lockless shared rings. While this setup delivers the best possible performance with standard AF_XDP in userspace, it lacks key features of FLASH, such as strong isolation, modularity, and ease of development, as all NFs are tightly coupled within a single process. In **AF_XDP-SRIOV**, chaining is facilitated through NIC-assisted Virtual Functions (VFs). **OpenNetVM** is a state-of-the-art DPDK-based NF chaining framework [45].

Setup. Experiments were conducted on two servers, each with a 24-core Intel Xeon Gold 5418Y CPU and 128GB RAM. We used two NIC configurations: Intel E810 100Gbps (ice driver) and Mellanox ConnectX-4 MT27700 40Gbps (mlx5 driver), the latter required for evaluating AF_XDP with SR-IOV, which is only supported on mlx5. Traffic was generated using Pktgen [42]. The system under test (SUT) used the same NIC type as the traffic generator and ran Ubuntu 22.04.5 with Linux kernel 6.10.6, with hyper-threading disabled. FLASH experiments were run on our modified kernel, while all baseline configurations used the stock Linux kernel.

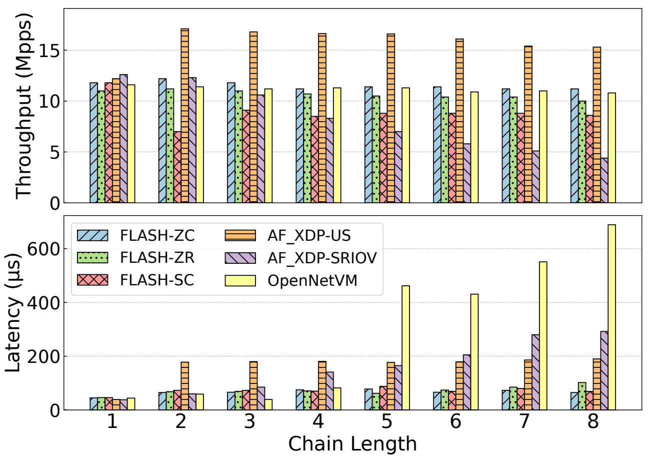

Figure 10: Throughput and latency comparison of NF chaining solutions with varying chain lengths (64B packet size).

6.1 NF-chaining performance with FLASH

We evaluate FLASH by measuring throughput and latency while varying the number of L2fwd NFs in a chain, each performing a MAC address update and redirection on 64-byte packets. To ensure a fair comparison with baselines, we modify MAC addresses in FLASH as well, even though the redirection mechanism does not require it. Figure 10 compares zero-copy (FLASH-ZC, FLASH-ZR) and single-copy (FLASH-SC) designs with the three baselines.

As shown in the figure, AF_XDP-US achieves the highest throughput among all configurations. This setup chains multiple NFs within a single userspace process, performing packet redirection through lockless shared memory. Only the first and last NFs in the chain communicate using AF_XDP system calls, leading to lower overheads. In contrast, FLASH maintains strong isolation by treating each NF as an independent unit with its own receive and transmit path. While FLASH has a 3–33% throughput gap compared to AF_XDP-US, it provides a more flexible and secure chaining model, suitable for multi-tenant and untrusted environments.

Compared to OpenNetVM, FLASH achieves comparable throughput (1×–1.07×) while being significantly more resource efficient, because OpenNetVM requires at least three dedicated CPU cores even for a single NF, whereas FLASH scales CPU usage based on the number of active sockets. FLASH significantly outperforms AF_XDP-SRIOV, with 1.1× to 2.5× higher throughput for chain lengths more than two. This aligns with prior findings [1, 19] on SR-IOV's poor performance due to IOTLB cache misses and VF-to-memory copy overhead. FLASH-ZR achieves up to 1.6× better throughput than FLASH-SC, benefiting from zero-copy redirection.

FLASH's throughput at a chain length of one is slightly lower than that of AF_XDP-SRIOV because the costs of packet redirection and smart scheduling are amortized when chaining multiple NFs. Similarly, the performance of Rust-based zero-copy FLASH-ZR is slightly lower than that of single-copy FLASH-SC in this case, because the packet copy overhead of FLASH-SC is absent, while the Rust-based safety checks in FLASH-ZR add a slight overhead.

In terms of latency, FLASH consistently performs best—up to 66% lower than AF_XDP-US, 78% lower than AF_XDP-SRIOV, and 91% lower than OpenNetVM for chains up to eight NFs. AF_XDP-US

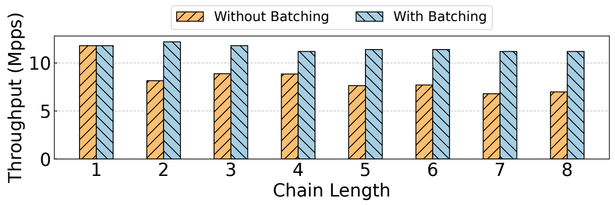

Figure 11: Throughput with and without batching optimization in FLASH-ZC.

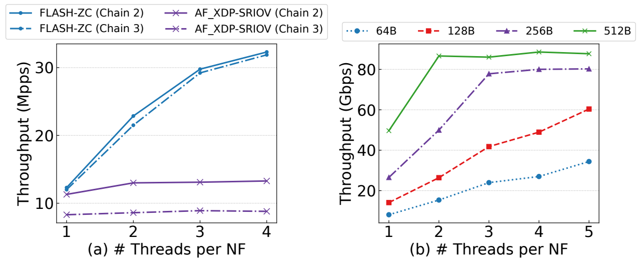

Figure 12: (a) Comparison of scaling of NFs by increasing # threads used per NF. (b) Scaling of a real-world NF chain of length four to 100 Gbps with different packet sizes.

incurs higher latency because its userspace thread must simultaneously monitor both redirection queues and AF_XDP queues, all within a single thread. While introducing multithreading could reduce this latency, it would necessitate locks in the data path to access the rings of AF_XDP and require busy polling across multiple threads, which adds complexity and increased overhead. OpenNetVM, on the other hand, suffers from contention at its centralized NF Manager. AF_XDP-SRIOV also exhibits high latency from DMA and IOTLB inefficiencies in longer chains.

6.2 Impact of FLASH optimisations

To evaluate the effectiveness of our design choices, we conducted a series of experiments aimed at isolating and analyzing the impact of individual features. We used AF_XDP-SRIOV as the baseline, as it offers comparable functionality, such as packet and memory isolation, while retaining kernel-native features through AF_XDP. This also enables the reuse of the same application and ensures a fair comparison with similar system resource requirements.

Batching. To understand the importance of batching, we measured the throughput of NF chains with and without batching, across varying chain lengths, where packets are redirected between multiple NFs. Figure 11 shows the results using chains of L2fwd NFs. Without batching, performance degrades significantly as the number of NFs in the chain increases, primarily due to the cost of frequent atomic operations. By incorporating batching, we reduce the frequency of these operations, enabling our system to deliver higher and more stable throughput, even as the chain length grows.

Scaling. Next, we evaluate the scalability of FLASH compared to AF_XDP-SRIOV by running chains of L2fwd NFs while increasing the number of threads allocated to each NF. In this setup, multiple parallel instances of the L2fwd NF are chained together, with each thread pinned to a separate CPU core. Figure 12a shows that FLASH-ZC demonstrates near-linear scalability with the number

	Individual NF Throughput	AF_XDP-SRIOV		FLASH	
		Service Rate	CPU	Service Rate	CPU
NF1	8.3 Mpps	8.3 Mpps	100%	4.28 Mpps	91%
NF2	9.3 Mpps	8.3 Mpps	100%	4.28 Mpps	52%
NF3	6.05 Mpps	6.05 Mpps	100%	4.28 Mpps	62%
NF4	4.36 Mpps	4.22 Mpps	100%	4.28 Mpps	100%
Final	-	4.22 Mpps	400%	4.28 Mpps	205%

Table 1: Throughput and CPU utilization.

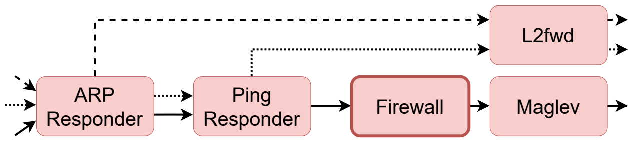

Figure 13: Five-NF Chain Setup—ARP and Ping responders, Firewall, Maglev, and L2fwd. Each of the NFs is running on a separate CPU core, and the Firewall is the bottleneck NF.

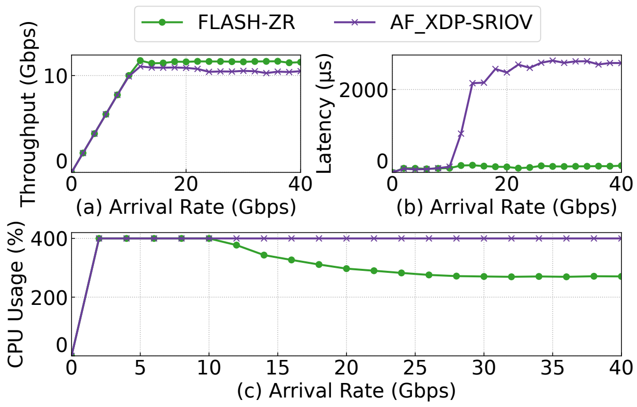

Figure 14: Arrival Rate vs. Throughput, Latency and CPU usage for a simple real world NF chain where an NF is a bottleneck.

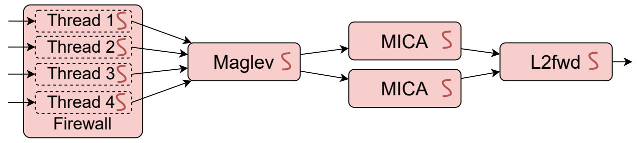

Figure 15: A complex NF chain with multi-threading, multiple ingress and egress points to and from the NFs.

of threads, up to the point where the NIC's bandwidth becomes the limiting factor. This indicates that FLASH-ZC can efficiently utilize available CPU and NIC resources without introducing bottlenecks. In contrast, in AF_XDP-SRIOV with scaling, the achievable throughput flattens due to increased IOTLB misses and hardware redirection limits and decreases as the chain length increases.

We further assess FLASH's scalability for a realistic NF chain using a 100 Gbps testbed. In this experiment, we deploy a chain of four lightweight NFs—ARP Responder, Ping Responder, Maglev, and L2fwd. Now, to scale, we increase the number of threads per NF and create parallel chains by linking corresponding threads across NFs. As shown in Figure 12b, FLASH maintains near-linear scalability, achieving up to 87 Gbps with 512B packets. While further scaling is possible, our evaluation was constrained by CPU availability. These results highlight FLASH's ability to efficiently exploit parallelism and deliver high throughput, approaching line-rate performance.

Early backpressure detection based smart polling. To evaluate FLASH's backpressure mechanism, we deployed a 4-NF chain with heterogeneous throughput capacities, as summarized in Table 1. The backpressure threshold was set to half the TX ring size, with a sleep interval of 1ms. We generated traffic at 40Gbps and measured both throughput and CPU utilization under this load. As seen in the table, both FLASH and AF_XDP-SRIOV achieved comparable end-to-end throughput of 4.28Mpps and 4.22Mpps, respectively, primarily limited by NF4, whose processing capacity was significantly lower than FLASH or AF_XDP-SRIOV. However, FLASH significantly reduces CPU usage, lowering it by up to 51% by throttling the first NF's packet reception to align with the capacity of downstream NFs. In contrast, AF_XDP-SRIOV kept all NFs running at full load, leading to unnecessary CPU consumption. With FLASH, each NF's CPU usage matched closely with its actual processing load, except for the first NF, which remained relatively active due to frequent interrupts. Without a backpressure mechanism, all NFs would consume 100% CPU, even while dropping excess packets.

6.3 Using FLASH with Real World NFs

We begin by evaluating a deployment consisting of five real-world network functions (NFs) with varying levels of complexity. The possible chain configurations within this deployment are illustrated in Figure 13. For this experiment, we generate traffic, ensuring all packets reach the firewall after being processed by the first two NFs. At the firewall, each incoming packet is processed against a set of 1,000 filtering rules evaluated sequentially. These rules are configured to permit all packets, forwarding them to the Maglev load balancer. This setup makes the firewall the computational bottleneck in the chain. As a result, the firewall limits overall performance to 2.72 Mpps or 11.38 Gbps for 512-byte packets, which is significantly lower than the capacity of FLASH or AF_XDP-SRIOV.

Now we measure the throughput, latency for the chain, and CPU usage for all the NFs with varying arrival rates. As illustrated in Figure 14c, when the system is idle, no CPU resources are consumed, and our implementation waits for incoming packets. Upon packet arrival, hardware interrupts trigger a transition to busy polling mode across all NFs. Initially, throughput matches the incoming packet rate as seen in Figure 14a. However, once the firewall becomes a bottleneck at around 11.38 Gbps, upstream NFs (ARP and ping responders) begin throttling. This throttling results in a gradual decrease in CPU utilization, as the first NF stops receiving packets. That is, not all interrupts trigger busy polling, and if any packets are received, they traverse through the chain smoothly. Importantly, the throttling does not impact the throughput or the latency. We expect that our in-progress blocking implementation will yield even greater CPU efficiency.

In contrast, an SR-IOV-based implementation, while capable of waking NFs via interrupt-based busy polling, fails to apply effective backpressure. This results in unnecessary CPU consumption and high packet latency (Figure 14b). FLASH provides at least 94% less latency and saves around 32% wasted CPU cycles at high loads. The maximum latency observed using AF_XDP-SRIOV was 2.7 ms. We suspect the high latency is due to excessive buffering in the NIC, as packets may be dropped at any point in the chain.

NF	Individual NF Throughput per thread	# Threads	Service Rate	
			AF_XDP-SRIOV	FLASH
NF1 - Firewall (Rust)	2.72 Mpps	4	5.98 Mpps	9.4 Mpps
NF2 - Maglev (Rust)	11 Mpps	1	5.33 Mpps	9.4 Mpps
NF3 - MICA (C)	8.8 Mpps	1	2.57 Mpps	4.7 Mpps
NF4 - MICA (C)	8.8 Mpps	1	2.57 Mpps	4.7 Mpps
NF5 - L2fwd (C)	11.8 Mpps	1	5.05 Mpps	9.4 Mpps
Aggregate	-	8	5.05 Mpps	9.4 Mpps

Table 2: Performance of complex real-world workload.

In the previous experiment, we showed that when NFs are deployed without scaling and the overall chain capacity is lower than the capacity of FLASH or AF_XDP-SRIOV, both systems achieve similar throughput. However, FLASH outperforms in terms of latency and CPU efficiency. We now shift focus to a more optimized deployment in which bottlenecks have been addressed through scaling. The updated NF chain, illustrated in Figure 15, represents a more complex and realistic setup. In this topology, some NFs are single-threaded while others are scaled across multiple threads. Specifically, the chain begins with a firewall (NF1) that applies 1,000 filtering rules evaluated sequentially for each packet. All rules allow traffic through to maintain a consistent workload without drops. Packets then flow to Maglev (NF2), a load balancer that evenly distributes traffic between two MICA instances (NF3 and NF4), which serve as in-memory key-value stores processing read-only queries, emulating a typical caching workload. Finally, packets are forwarded to L2fwd (NF5), which returns them to the client. This setup forms a complex NF graph featuring both fan-out (Maglev to MICA) and fan-in (MICA to L2fwd) patterns, demonstrating FLASH's capability to handle arbitrary NF topologies. The firewall (NF1), originally limited to 2.72 Mpps, was scaled using four threads pinned to four separate CPU cores to eliminate the bottleneck.

Table 2 summarizes the capacity and throughput achieved in both FLASH and AF_XDP-SRIOV. Despite the chain being composed of diverse components implemented in multiple languages, FLASH maintained high performance and flexibility. In this setup, the primary bottleneck in FLASH was Maglev (NF2), yet it still achieved 1.86× higher throughput than AF_XDP-SRIOV. The end-to-end throughput reached 9.4 Mpps, slightly below the peak due to the overhead of distributing traffic to multiple downstream NFs.

In contrast, AF_XDP-SRIOV is bottlenecked at L2fwd (NF5), causing upstream NFs to continue processing at full rate, leading to queue buildup and wasted CPU cycles. This experiment highlights FLASH's ability to scale individual NFs independently and efficiently support complex and realistic chaining patterns, making it well-suited for practical NF deployments.

6.4 Overhead Analysis

We evaluated the performance of FLASH's modified ring design by comparing its throughput and latency against the default SPSC rings of AF_XDP in different modes. As shown in Figure 16, both designs deliver nearly identical results across both metrics. This demonstrates that the additional logic introduced to support flexible chaining incurs negligible overhead. Overall, FLASH's ring design preserves the efficiency of the default implementation while enabling greater flexibility and extended functionality.

Since FLASH supports complex NF chains with arbitrary in-degrees and out-degrees, it's important to evaluate the overhead introduced by such topologies, enabled by our modifications to AF_XDP rings. Figure 17a shows Maglev's throughput as out-degree increases under scaled conditions. As expected, redirecting packets to multiple NFs adds overhead, leading to a noticeable throughput drop as Maglev scales with more threads.

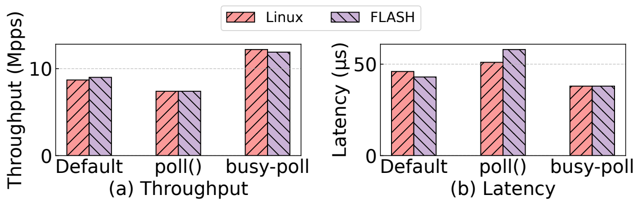

Figure 16: Performance of SPSC vs MPSC/SPMC.

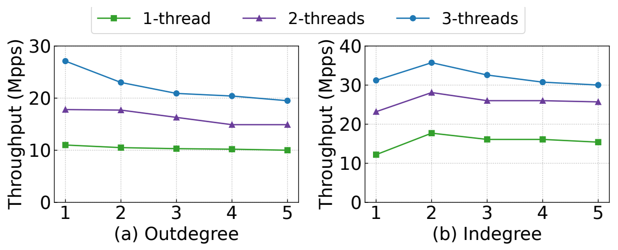

Figure 17: Throughput with varying in/out-degree.

Figure 17b shows throughput when increasing the in-degree by forwarding packets from multiple upstream NFs to a single downstream NF. We observe that with an in-degree of two, throughput initially improves. This is because, in a typical NF chain, the first `napi_poll` handles both packet reception from the NIC and redirection, becoming a bottleneck, while subsequent `napi_poll` instances remain underutilized. So when the in-degree increases, the CPU is properly utilized. However, further increases in in-degree lead to contention at the receiving NF, where concurrent enqueues from multiple threads activate multi-producer synchronization, adding overhead and reducing performance.

We conducted experiments in both bare-metal and containerized deployments and observed similar results in terms of performance. This is due to no additional overhead in the network path, while still providing strong isolation using in-kernel packet redirection.

7 Conclusions & Future Work

We presented FLASH, an extension to the Linux AF_XDP subsystem for high-performance, kernel-native chaining of co-located NFs. FLASH supports zero-copy chaining for performance and single-copy mode for compatibility with legacy AF_XDP applications. It enables efficient packet redirection through kernel-level enhancements, including multi-producer/consumer AF_XDP rings and optimized thread wakeup mechanisms. Our experiments show that FLASH delivers throughput comparable to kernel bypass frameworks, while offering greater efficiency, low latency, flexibility, and improved support for multi-tenant cloud environments. As part of future work, FLASH will extend its capabilities to handle per-flow backpressure and support dynamic scheduling of network functions to eliminate reliance on direct busy-polling. In addition, we plan to extend our FLASH library to support transport-layer applications, enabling features such as packet reassembly.

References

[1] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. 2010. IOMMU: strategies for mitigating the IOTLB bottleneck. In *ISCA '10*. Springer-Verlag, Berlin, Heidelberg, 256–274. https://doi.org/10.1007/978-3-642-24322-6_22

[2] Fabien André, Stéphane Gouache, Nicolas Le Scouarnec, and Antoine Monsifrot. 2018. Don't share, Don't lock: Large-scale Software Connection Tracking with Krononat. In *ATC '18*. USENIX Association, Boston, MA, 453–466. https://www.usenix.org/conference/atc18/presentation/andre

[3] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding host network stack overheads. In *SIGCOMM '21*. Association for Computing Machinery, New York, NY, USA, 65–77. https://doi.org/10.1145/3452296.3472888

[4] Intel Corporation. 2018. XDP user-space ring structure. Retrieved October 20, 2025 from https://github.com/torvalds/linux/blob/master/net/xdp/xsk_queue.h

[5] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, and Dima Rubinstein et al. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *NSDI '18*. USENIX Association, Renton, WA, 373–387. https://www.usenix.org/conference/nsdi18/presentation/dalton

[6] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. 2012. Toward predictable performance in software packet-processing platforms. In *NSDI '12*. USENIX Association, USA, 11. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/dobrescu

[7] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. 2009. RouteBricks: exploiting parallelism to scale software routers. In *SOSP '09*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/1629575.1629578

[8] DPDK. 2025. Multi-process Limitations, Multi-process Support, DPDK. Retrieved October 20, 2025 from https://doc.dpdk.org/guides/prog_guide/multi_proc_support.html#multi-process-limitations

[9] DPDK. 2025. Ring Library, DPDK. Retrieved October 20, 2025 from https://doc.dpdk.org/guides/prog_guide/ring_lib.html

[10] dpdk.org. 2025. Data Plane Development Kit. Retrieved October 20, 2025 from https://www.dpdk.org/

[11] Danielle E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *NSDI '16*. USENIX Association, Santa Clara, CA, 523–535. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud

[12] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator. In *IMC '15*. Association for Computing Machinery, New York, NY, USA, 275–287. https://doi.org/10.1145/2815675.2815692

[13] Xincai Fei, Fangming Liu, Hai Jin, and Bo Li. 2020. FlexNFV: Flexible Network Service Chaining with Dynamic Scaling. *IEEE Network* 34, 4 (2020), 203–209. https://doi.org/10.1109/MNET.001.1900483

[14] Matt Fleming. 2017. A thorough introduction to eBPF. Retrieved October 20, 2025 from https://lwn.net/Articles/740157/

[15] FreeBSD. 2009. bufring in FreeBSD v8. Retrieved October 20, 2025 from https://svnweb.freebsd.org/base/release/8.0.0/sys/sys/buf_ring.h?revision=199625&amp;view=markup

[16] Tom Herbert and Willem de Bruijn. 2019. Scaling in the Linux Networking Stack. Retrieved October 20, 2025 from https://docs.kernel.org/networking/scaling.html

[17] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The eXpress data path: fast programmable packet processing in the operating system kernel. In *CoNEXT '18*. Association for Computing Machinery, New York, NY, USA, 54–66. https://doi.org/10.1145/3281411.3281443

[18] David Howells and Paul E. McKenney. 2010. Circular Buffers. Retrieved October 20, 2025 from https://docs.kernel.org/core-api/circular-buffers.html

[19] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2014. NetVM: high performance and flexible networking using virtualization on commodity platforms. In *NSDI '14*. USENIX Association, USA, 445–458. https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-hwang.pdf

[20] Panpan Jin, Xincai Fei, Qixia Zhang, Fangming Liu, and Bo Li. 2020. Latency-aware VNF Chain Deployment with Efficient Resource Reuse at Network Edge. In *IEEE INFOCOM '20*. Institute of Electrical and Electronics Engineers, Virtual Conference, 267–276. https://doi.org/10.1109/INFOCOM41043.2020.9155345

[21] Magnus Karlsson. 2020. xsk: support shared umems between devices and queues. Retrieved October 20, 2025 from https://lore.kernel.org/netdev/1598603189-32145-1-git-send-email-magnus.karlsson@intel.com/#r

[22] Magnus Karlsson, Gary Loughnane, Elza Mathew, Paulina Osikoya, Jeff Shaw, Maryam Tahhan, Edwin Verplanke, and Keith Wiles. 2022. Cloud Native Data Plane (CNDP) - Overview. Retrieved October 20, 2025 from https://builders.intel.com/solutionslibrary/cloud-native-data-plane-cndp-overview-technology-guide

[23] Magnus Karlsson and Bjorn Topel. 2018. The Path to DPDK Speeds for AF XDP. In *LPC '18*. Linux Plumbers Conference, Vancouver, BC, 1–9. https://lpc.events/event/2/contributions/99/

[24] Sameer G Kulkarni, Guyue Liu, K. K. Ramakrishnan, Mayutan Arumaithurai, Timothy Wood, and Xiaoming Fu. 2018. REINFORCE: achieving efficient failure resiliency for network function virtualization based services. In *CoNEXT '18*. Association for Computing Machinery, New York, NY, USA, 41–53. https://doi.org/10.1145/3281411.3281441

[25] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. 2017. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *SIGCOMM '17*. Association for Computing Machinery, New York, NY, USA, 71–84. https://doi.org/10.1145/3098822.3098828

[26] Hao Li, Yihan Dang, Guangda Sun, Guyue Liu, Danfeng Shan, and Peng Zhang. 2023. LemonNFV: Consolidating Heterogeneous Network Functions at Line Speed. In *NSDI '23*. USENIX Association, Boston, MA, 1451–1468. https://www.usenix.org/conference/nsdi23/presentation/li-hao

[27] Xiaoyao Li, Xiuxiu Wang, Fangming Liu, and Hong Xu. 2018. DHL: Enabling Flexible Software Network Functions with FPGA Acceleration. In *IEEE ICDCS '18*. Association for Computing Machinery, Vienna, Austria, 1–11. https://doi.org/10.1109/ICDCS.2018.00011

[28] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: a holistic approach to fast in-memory key-value storage. In *NSDI '14*. USENIX Association, USA, 429–444. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim

[29] Linux manual page. 2024. unix - sockets for local interprocess communication. Retrieved October 20, 2025 from https://man7.org/linux/man-pages/man7/unix.7.html

[30] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the art of network function virtualization. In *NSDI '14*. USENIX Association, USA, 459–473. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins

[31] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. 2018. Creating Complex Network Services with eBPF: Experience and Lessons Learned. In *HPSR '18*. Institute of Electrical and Electronics Engineers, Bucharest, Romania, 1–8. https://doi.org/10.1109/HPSR.2018.8850758

[32] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu. 2021. A Framework for eBPF-Based Network Functions in an Era of Microservices. *IEEE Trans. on Netw. and Serv. Manag.* 18, 1 (2021), 133–151. https://doi.org/10.1109/TNSM.2021.3055676

[33] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *OSDI '16*. USENIX Association, Savannah, GA, 203–216. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda

[34] Federico Parola, Roberto Procopio, Roberto Querio, and Fulvio Risso. 2023. Comparing User Space and In-Kernel Packet Processing for Edge Data Centers. *SIGCOMM CCR* 53, 1 (2023), 14–29. https://doi.org/10.1145/3594255.3594257

[35] Federico Parola, Shixiong Qi, Anvaya B. Narappa, K. K. Ramakrishnan, and Fulvio Risso. 2024. SURE: Secure Unikernels Make Serverless Computing Rapid and Efficient. In *SoCC '24*. Association for Computing Machinery, New York, NY, USA, 668–688. https://doi.org/10.1145/3698038.3698558

[36] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. 2022. SPRIGHT: extracting the server from serverless computing! high-performance eBPF-based event-driven, shared-memory processing. In *SIGCOMM '22*. Association for Computing Machinery, New York, NY, USA, 780–794. https://doi.org/10.1145/3544216.3544259

[37] Shixiong Qi, Songyu Zhang, K. K. Ramakrishnan, Diman Zad Tootaghaj, Hardik Soni, and Puneet Sharma. 2025. Palladium: A DPU-enabled Multi-Tenant Serverless Cloud over Zero-copy Multi-node RDMA Fabrics. In *SIGCOMM '25*. Association for Computing Machinery, New York, NY, USA, 1257–1259. https://doi.org/10.1145/3718958.3750494

[38] redhat. 2025. afxdp plugins for kubernetes. Retrieved October 20, 2025 from https://github.com/redhat-et/afxdp-plugins-for-kubernetes

[39] Nishanth Shyamkumar, Piotr Raczynski, Dave Cremins, Michal Kubiak, and Ashok Sunder Rajan. 2022. In-Kernel Fast Path Performance For Containers Running Telecom Workloads. In *Netdev 0x16*. netdev conference, Lisbon, Portugal, 1–6. https://netdevconf.info/0x16/sessions/talk/inkernel-fast-path-performance-for-containers-running-telecom-workload.html

[40] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. 2021. revisiting the open vSwitch dataplane ten years later. In *SIGCOMM '21*. Association for Computing Machinery, New York, NY, USA, 245–257. https://doi.org/10.1145/3452296.3472914

[41] Matthew Wilcox. 2018. Linux ID Allocation. Retrieved October 20, 2025 from https://docs.kernel.org/core-api/idr.html

[42] Keith Wiles. 2014. Pktgen—Traffic generator powered by DPDK. Retrieved October 20, 2025 from https://github.com/pktgen/Pktgen-DPDK

[43] Yikai Xiao, Qixia Zhang, Fangming Liu, Jia Wang, Miao Zhao, Zhongxing Zhang, and Jiaxing Zhang. 2019. NFVdeep: Adaptive Online Service Function

Chain Deployment with Deep Reinforcement Learning. In *IWQoS '19*. Institute of Electrical and Electronics Engineers, Phoenix, AZ, USA, 1–10. https://doi.org/10.1145/3326285.3329056

[44] Ziteng Zeng, Leslie Monis, Shixiong Qi, and K. K. Ramakrishnan. 2022. MiddleNet: A High-Performance, Lightweight, Unified NFV and Middlebox Framework. In *NetSoft '22*. Institute of Electrical and Electronics Engineers, Milan, Italy, 180–188. https://doi.org/10.1109/NetSoft54395.2022.9844083

[45] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. 2016. OpenNetVM: A Platform for High Performance Network Service Chains. In *HotMIddlebox '16*. Association for Computing Machinery, New York, NY, USA, 26–31. https://doi.org/10.1145/2940147.2940155