Comparing User Space and In-Kernel Packet Processing for Edge Data Centers

Original
Comparing User Space and In-Kernel Packet Processing for Edge Data Centers / Parola, Federico; Procopio, Roberto; Querio, Roberto; Risso, Fulvio. - In: COMPUTER COMMUNICATION REVIEW. - ISSN 0146-4833. - ELETTRONICO. - 53:1(2023), pp. 14-29. [10.1145/3594255.3594257]

Availability:
This version is available at: 11583/2978249 since: 2023-04-30T23:47:46Z

Publisher:
ACM

Published
DOI:10.1145/3594255.3594257

Terms of use:
This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
ACM postprint/Author's Accepted Manuscript

© ACM 2023. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in COMPUTER COMMUNICATION REVIEW, http://dx.doi.org/10.1145/3594255.3594257.

(Article begins on next page)
Comparing User Space and In-Kernel Packet Processing for Edge Data Centers

Federico Parola
Politecnico di Torino, Italy
federico.parola@polito.it

Roberto Querio
TIM S.p.A., Italy
roberto.querio@telecomitalia.it

Roberto Procopio
TIM S.p.A., Italy
roberto.procopio@telecomitalia.it

Fulvio Risso
Politecnico di Torino, Italy
fulvio.risso@polito.it

ABSTRACT
Telecommunication operators are massively moving their network functions in small data centers at the edge of the network, which are becoming increasingly common. However, the high performance provided by commonly used technologies for data plane processing such as DPDK, based on kernel-bypass primitives, comes at the cost of rigid resource partitioning. This is unsuitable for edge data centers, in which efficiency demands both general-purpose applications and data-plane telco workloads to be executed on the same (shared) physical machines. In this respect, eBPF/XDP looks a more appealing solution, thanks to its capability to process packets in the kernel, achieving a higher level of integration with non-data plane applications albeit with lower performance than DPDK. In this paper we leverage the recent introduction of AF_XDP, an XDP-based technology that allows to efficiently steer packets in user space, to provide a thorough comparison of user space vs in-kernel packet processing in typical scenarios of a data center at the edge of the network. Our results provide useful insights on how to select and combine these technologies in order to improve overall throughput and optimize resource usage.

CCS CONCEPTS
• Networks → Middle boxes / network appliances; Network servers; Programmable networks; • Software and its engineering → Communications management;

KEYWORDS
Data Plane, XDP, AF_XDP, eBPF, NFV, Linux Kernel

1 INTRODUCTION
Edge computing is a paradigm that moves computational capabilities close to the end user, in order to provide services with lower latency and increase the amount of available bandwidth. Unlike cloud computing, where user data is processed in big, centralized data centers with almost unlimited resources, processing at the edge requires telco operators to support a large number of small, distributed data centers, each one featuring a few servers. Traffic of the user reaching these data centers has to be processed by a fixed chain of Network Functions (NFs) that provide basic connectivity to the global network (e.g., 5G User Plane Functions, a.k.a. UPFs, NATs). This traffic might be further processed by additional network services (e.g., firewalls) and can be either directed to the Internet or to applications running in the same data center (e.g., 5G control plane services, object recognition software, content caches, etc.), located at the edge for different reasons such as latency requirements, data aggregation, or resiliency concerns.

Traditionally, the above two types of workload (data plane NFs and traditional applications) are handled by partitioning available servers in two subsets. Data-plane workloads are usually executed on servers that leverage kernel-bypass packet processing frameworks such as Intel DPDK. Instead, traditional applications are orchestrated by platforms such as Kubernetes and leverage the widespread Linux TCP/IP networking stack. In fact, kernel-bypass technologies allow to process packets in user space, completely avoiding the overheads introduced by the kernel network stack. They are well known for their flexibility and for providing very high throughput, but can hardly be executed in servers running traditional applications due to the necessity to rely on rigid resource allocation schemes (e.g., CPU pinning with dedicated CPU cores, huge memory pages), and the difficulties to support applications that leverage the standard TCP/IP stack (which, in fact, needs to be re-implemented in user space [12]). On the other side, existing kernel-level network processing primitives (e.g., Netfilter, Traffic Control, etc.) are highly integrated with applications relying on the network stack, but introduce an unnecessary overhead (hence, low throughput) to pass-through traffic, that only needs NF processing.

While the approach based on rigid servers partitioning may be appropriate in cloud data centers, it may provide a sub-optimal resource usage when a few servers are available, which could severely impact the edge scenario. On one side, packets moving between NF and application servers generate additional traffic in the data center (east-west traffic) and require additional I/O operations. On the other side, the rigid partitioning of servers based on the type of application may lead to wasting some of the available resources. In this scenario, a shared approach that consolidates both workloads on the same machine(s) would enable a more efficient usage of resources, allowing to co-locate NFs and applications working on the same traffic and to allocate spare resources to any kind of workload.

In recent years, the introduction of the eXpress Data Path (XDP) [11] and AF_XDP has provided the missing ingredients to efficiently handle traffic either in kernel or in user space, on the same platform. XDP allows to process packets in the NIC driver [6, 19], retaining the possibility to yield a packet to the Linux network stack, while AF_XDP sockets can be used to bypass limitations of eBPF programs and provide flexible processing in user space. However, it is still unclear how to use the above technologies at the same time in a
single server, to handle the complex processing scenario envisioned at the edge of the network.

This paper presents the performance analysis of in-kernel and user space packet processing based on XDP/AF_XDP, including pass-through traffic, processed by a chain of NFs and redirected to a remote destination, local traffic, directed to applications running locally, and dropped traffic, which has to be discarded e.g., for security reasons. This has the aim of determining which technology is best suited for each case, and to provide insights on how to optimally handle the mixed scenario typical of edge data centers.

While most recent network cards, such as Intel 800 series, provide customized hardware packet processing at the NIC level, we did not include this technology in our evaluation. Indeed, this paper focuses on a fully software approach, which allows to be completely independent from the underlying hardware and supports the case (rather common at the time of writing) of the many data centers that feature network cards with limited packet processing capabilities (such as our Intel 700 series).

This paper is structured as follows. Section 2 provides a background on the two main technologies considered in this paper (namely, XDP and AF_XDP), as well as on HW/SW packet steering mechanisms that proved to be a key to further improve performance. Section 3 details the methodology and scenarios encompassed in our tests, while Sections 4 to 6 present the results of our experiments for the above tests scenarios, namely dropped, pass-through and local traffic. Section 7 combines the takeaways from previous experiments to provide guidelines to handle heterogeneous combinations of traffic and experimentally verifies their effectiveness. Section 8 reviews the literature related to the topic, while Section 9 draws the main conclusions.

2 BACKGROUND

2.1 eBPF/XDP

eBPF is a virtual machine that allows the extensions of the functionalities of the kernel with custom code that can be injected at run time and executed at various hooks points (e.g. trace points, system calls, every kernel function, etc.). eBPF programs leverage a special bytecode that is generated by the Clang/LLVM toolchain starting from a source code written in a (restricted) C language, which can be compiled just-in-time into native machine code for extra performance. Upon injection, eBPF programs are analyzed by a verifier, whose aim is to guarantee that the code cannot harm the kernel, for example checking that only allowed memory accesses are performed and that the program will eventually terminate. As a consequence, eBPF programs have some limitations, such as a maximum number of instructions and the lack of support for unbounded loops. Moreover, they cannot access memory in a custom way, but need to rely on maps, a set of key-value stores with different access semantics (array, hash, queue, etc.), that can be shared between several eBPF programs and with the user space, and can be used to preserve the state among multiple executions of a program. Despite these limitations, eBPF has proven to be suitable to the creation of reasonably complex NFs, especially if limited to headers processing [18].

The eXpress Data Path provides a hook to execute high speed eBPF packet processing programs before actually entering the main Linux kernel. XDP allows, in its native mode, to execute these programs in the NIC driver, at the earliest possible point after a packet is received from the hardware, before the kernel allocates its per-packet sk_buff data structure or performs any parsing. Based upon the result of the processing, the program can either ask the kernel to drop the packet, let it continue for further processing in the network stack, send it back on the receiving interface or redirect it to another net device or to user space thanks to AF_XDP sockets.

2.2 AF_XDP sockets

AF_XDP sockets are a new socket family that allows user space code to exchange packets with the NIC with very limited overhead, very similar to existing kernel-bypass technologies [5]. An application leveraging AF_XDP has to create an array of user-space memory called UMEM. The UMEM is a chunk of contiguous memory divided into equal-sized buffers, each one holding a single Ethernet frame. These buffers are used to move packets between the NIC driver and the user space application, exchanging their pointers through four circular rings allocated by the kernel. Figure 1 shows the lifecycle of a UMEM buffer in an application receiving packets from the network and dropping/redirecting them. At startup, buffers are waiting in the f1111 ring (1). When a packet arrives, the NIC stores it in a buffer available in the f1111 ring and moves its pointer to the rx ring (2), where it is consumed (i.e., processed) by the user space application (3). If the packet is dropped, its buffer is re-added to the f1111 ring (3a); otherwise it is queued to the tx ring for redirection (3b). The driver transmits packets from the tx ring and moves their buffers to the comp (i.e., completion) ring (4), leaving to the user space application the responsibility to move the above buffers back to the f1111 ring (5), ready to keep new packets.

Every AF_XDP socket is bound to a single {netdev, queue} couple (multiple sockets for the same couple are allowed), and is associated with one rx and one tx ring. A single UMEM can be shared between different sockets, however, a new pair of f1111 and comp rings is needed for every {netdev, queue} couple handled. This allocation of rings guarantees a single-producer single-consumer access pattern on the kernel side, allowing faster operations. In user-space, the responsibility to make sure that no concurrent access to the same ring can occur is left to the programmer. When a packet is received by the NIC, it is first processed by an XDP packet processing program, which can be compiled just-in-time into native machine code for ACM SIGCOMM Computer Communication Review

Volume 53 Issue 1, January 2023
program, that can choose to redirect it towards an AF_XDP socket stored into a map of type XSKMAP. In the initial implementation of AF_XDP, packets where first DMAed by the NIC into a kernel-owned XDP buffer, then copied into a UMEM buffer and sent to user space (a similar operation was performed for transmission). A zero-copy mode was later introduced [25], allowing the NIC to DMA packets directly into/from a UMEM buffer and move them to user space without expensive copies. While the copy mode works on all drivers supporting XDP, the zero-copy mode requires an explicit driver support.

In a standard AF_XDP-based packet processing program, the NIC triggers an interrupt after the reception of one or more packets. Packets are then processed by the driver poll function in the NAPI software interrupt context (softirq) and possibly moved to the AF_XDP rx ring, and the application has to check the ring for the presence of new packets (a similar process happens for transmission). This can be done either with a busy loop or with the poll() system call that puts the program into a wait state until buffers are available. The user space application and the driver can either be executed on the same core, with the consequent cost of continuous context switching between app and softirq, or on separated cores, this time adding the cost of realigning the caches of the different CPU cores (cache coherency). An additional preferred busy-polling mode was recently introduced in [26]. With this mode, interrupts are disabled and the user space application is responsible of periodically executing the driver poll function with a system call (sendto() or recvfrom()). This allows running the application and the driver on a single core eliminating all the context switching and coherency traffic costs, but at the cost of executing a syscall for each batch of transmitted/received packets.

### 2.3 Packet steering mechanisms

Given the massive number of processing cores available in modern CPUs, a key problem is how to distribute traffic load among the different available CPU cores, which is important for both NFs and traditional applications. This can be achieved leveraging either software-based techniques running in the kernel, or hardware-based mechanisms available on the NIC, such as the following.

#### 2.3.1 Receive Side Scaling (RSS) [16].

It allows a NIC to distribute incoming packets on multiple queues, which can then be processed by different CPU cores without contention (assuming no dependencies are present in the above traffic). RSS typically applies a hash function to the packet 5-tuple to identify the target queue, which guarantees that packets belonging to the same session are processed on the same core.

#### 2.3.2 Receive Packet Steering (RPS) [4].

It is a software implementation of RSS in Linux. Whereas RSS selects the queue and hence CPU that will run the hardware interrupt handler, RPS selects the CPU to perform protocol processing above the interrupt handler. This is accomplished by placing the packet on the desired CPU’s backlog queue and waking up the CPU for processing. Like in RSS, the target CPU is selected applying a hash function on the packet 5-tuple. Each receive hardware queue has an associated list of CPUs to which RPS may enqueue packets for processing.

#### 2.3.3 Receive Flow Steering (RFS) [7].

It is an extension of RPS that redirects packets to the core where the consuming application is running. This increases the performance by improving the cache locality for data structures handling the session (both kernel and user/space processing happens on the same core), and avoids copies of the packet among cores [20].

#### 2.3.4 Ntuple filters [16].

They define a set of rules, configured on the NIC hardware, that can (i) steer packets to a given queue, (ii) drop traffic or (iii) enforce specific hash options for RSS; this is called Ethernet Flow Director [21] on Intel NICs. Flow Director rules can be either inserted manually (Externally Programmed Mode) such as through ethtool, or automatically populated through the proprietary Application Targeting Routing technology (ATR).

#### 2.3.5 Application Targeting Routing (ATR) [21].

It represents the hardware acceleration of RSS available on selected Intel NICs: the NIC driver samples some of the outgoing packets and automatically generates hardware rules that force the incoming traffic to be sent to the same queue/core where the application is running.

### 3 BENCHMARKING METHODOLOGY

#### 3.1 Objectives

This section defines a benchmarking methodology for the performance characterization of XDP and AF_XDP, with the final objective of determining the best technology to be used on servers that host both traditional (i.e., computing intensive) and data plane (i.e., network intensive) workloads. For this aim, we identified three classes of traffic that must be handled by our server, each one characterized by a different processing path in the Linux networking stack. Dropped traffic refers to packets discarded by the NF, such as in case of a firewall, a DDoS mitigator or (partly) a traffic shaper. Pass-through traffic refers to packets that are forwarded to a remote destination after being processed by one or more NFs on the local server, such as in case of a load balancer redirecting packets towards backends running on different servers. Local traffic refers to packets that have to be processed by an application running on same server as the NF, e.g., traffic that is inspected by a firewall and is terminated on a local application, such as a Kubernetes pod running locally. Even though the above three scenarios are usually combined in a common deployment, we analyzed them in isolation to facilitate the profiling of the technologies under test, and then used results to determine their best combination in real use cases.

For each class of traffic we analyzed four cases. First, we evaluated raw I/O performance (Pure I/O), i.e., the impact of non-avoidable components such as the NIC driver and the basic XDP and AF_XDP mechanisms on the overall throughput, using a minimal program that simply swaps the MAC addresses of the packet. This test highlights the maximum theoretical throughput obtained by each technology, which can be considered our ideal target. Second, with respect to the dropped and pass-through test cases, we profiled the performance by running software with increasing complexity, either in terms of CPU demand (i.e., programs whose processing logic has different degrees of complexity) and Memory demand (i.e., different amount of RAM addressed by the application). The former aims at determining the impact of the data plane program complexity on the overall performance, while the
<table>
<thead>
<tr>
<th>Test mode</th>
<th>Packet processing location</th>
<th>User space packet notification mode</th>
<th>Driver execution mode</th>
<th>AF_XDP sockets enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>XDP</td>
<td>Kernel</td>
<td>-</td>
<td>Interrupt</td>
<td>No</td>
</tr>
<tr>
<td>AF_XDP</td>
<td>User</td>
<td>Busy loop</td>
<td>Interrupt</td>
<td>Yes</td>
</tr>
<tr>
<td>AF_XDP sysc</td>
<td>User</td>
<td>Busy loop</td>
<td>Syscall</td>
<td>Yes</td>
</tr>
<tr>
<td>AF_XDP poll</td>
<td>User</td>
<td>poll()</td>
<td>Interrupt</td>
<td>Yes</td>
</tr>
<tr>
<td>XDP-sk</td>
<td>Kernel</td>
<td>poll()</td>
<td>Interrupt</td>
<td>Yes</td>
</tr>
<tr>
<td>XDP-sk sysc</td>
<td>Kernel</td>
<td>Busy loop</td>
<td>Syscall</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1: Main characteristics of the encompassed test modes.

latter aims at determining the impact of the amount of allocated memory, which, surprisingly to us, is not orthogonal to the chosen processing technology. Processing complexity was measured by creating a program that recomputed the L4 checksum of the packet an increasing number of times. Memory demand was measured by allocating an array with increasing size and performing, for each packet, one random access in the above memory. In this respect, we verified how the generation of a random number introduces a minimal additional (but constant) CPU processing cost and has almost no memory impact. At the same time, random memory accesses make irrelevant the hardware pre-fetching capabilities of the CPU, hence avoiding the forecast of future requests and the masking of memory access costs. Finally, we ended our evaluation with a (proof-of-concept) real application (Traditional NF), to confirm that our findings are actually verified in a realistic scenario.

Our analysis focuses on the throughput of the technologies under test, albeit the latency represents another important metric, for the sake of space we leave its evaluation for a future work.

3.2 Benchmarked technologies

For in-kernel packet processing we executed our NFs in the standard XDP native mode (XDP in Table 1) to leverage all the advantages of early packet redirection/discarding.

For user space packet processing we executed our AF_XDP-based NFs in three different modes, all relying on the zero-copy user-kernel interaction. In standard mode (AF_XDP in Table 1) the NIC driver execution is triggered by the traditional interrupt/NAPI based mechanism; we performed a busy loop to check for the presence of new descriptors in AF_XDP rings in our user-space packet processing thread. In the system call mode (AF_XDP sysc in Table 1) we enabled the SO_PREFER_BUSY_POLLING flag to trigger the execution of the NIC driver through a system call executed in a (user-space) busy loop, leaving interrupts disabled, and configured the network interface as suggested in [26]. Instead, the poll mode (AF_XDP poll in Table 1) replaced the busy loop mechanism with the poll() system call, leaving the user space code in a waiting state until packets are received, all triggered by an interrupt.

Enabling AF_XDP sockets changes the way packet buffers are managed and how the code of the NIC driver handles packets, therefore impacting also XDP performance. Hence, we defined two combined test modes (XDP-sk and XDP-sk sysc in Table 1, the former relying on the traditional interrupt-based mechanism to trigger the NIC driver, the latter relying on a system call executed in a busy loop running in user space), in which AF_XDP sockets are enabled even if packets are completely processed at the XDP level and never reach user space.

All our tests (unless specified differently) run on a single CPU core, to prevent entering the multi-core scalability domain, whose study is left to a future work. While this configuration fits perfectly in-kernel processing, where a single processing context is scheduled (i.e., the NIC interrupt context), it may raise some concern in the user space case, in which both the interrupt and the user space application are potentially running at the same time. For example, [14] suggests to handle NIC interrupt requests (IRQ) and applications (hence, kernel vs user-space processing) on different CPU cores to avoid context switches and maximize performance. Nonetheless, we scheduled them on the same core to simplify the comparison with other technologies, paying attention not to affect the validity of our results. In fact, we always tuned the offered load to maximize the throughput of the system, achieving a result that was more than half the one obtained with two distinct CPU cores. This guarantees the fairness of our results, because our tuning avoids the livelock phenomenon in which the code running at the highest priority, i.e., the kernel, consumes most of the resources while the rest tends to starve, as shown in [13].

3.3 Testbed

Our testbed is composed of two servers equipped with a dual-port Intel XL710 40 Gbps NIC, only one port used, connected back-to-back. One server operates as Device Under Test (DUT) and the other one as tester/load generator. Both machines feature an Intel Xeon Gold 5120 14-cores CPU @ 2.20 GHz with Hyper-Threading and Turbo Boost disabled. The processor is provided with 32 KiB of per-core L1 data cache (corresponding to 512 x 64B lines), 1 MiB of per-core L2 cache (∼16K x 64B lines) and a 19.25 MiB unified L3 cache (∼315K x 64B lines). The servers run Ubuntu 20.04.4 LTS with kernel 5.14. The traffic is terminated on the DUT for the Dropped and Local tests, while is sent back to the traffic generator for the Pass-through tests. The DUT supports Intel DDIO technology, that allows the network card to DMA packets directly to/from the Last Level Cache (LLC, a.k.a. L3), hence avoiding high latencies due to the access to the main memory. As suggested in [8] (and confirmed in our tests), we increased the number of LLC ways available to Intel DDIO from 2 to 6 to improve the throughput of NFs and reduce packet losses.

1Notably, this does not apply to the AF_XDP sysc case in which IRQs and applications have to be scheduled on the same core to achieve decent performance; more in Section 2.2.

2sudo wrmsr 0xc8b 0x7e0.
For the dropped and pass-through test cases we used MoonGen as packet generator, generating minimum size UDP packets (64B Ethernet frames) towards the XDP/AF_XDP NF running on the DUT. We measured the throughput of the NPs according to RFC2544, tuning the input rate till the packet loss was lower than 0.1%. For local traffic tests we selected memcached [9] as a sample application, since it is likely to be deployed at the edge of the network and it is rather network intensive. We executed it with a variable number of threads and with a memory limit extended to 128MB (default is 64MB), and we pinned it to a set of cores (taskset command), either shared or disjoint from the ones used by the NF depending on the benchmarked technology (more details in Section 6). Requests on the tester machines were generated with Memoslap\textsuperscript{4} running on four cores, 128 I/O scenarios per core, each one establishing a TCP connection and requesting items with random keys for ten seconds. The sar tool was used to measure the CPU utilization of the DUT, split between user space (User), system calls (System) and software interrupt processing (SoftIRQ), the latter two both related to kernel space code. We also leveraged the perf tool to monitor the number of LLC hits and misses (the latter representing the number of memory accesses), since (i) memory access latency is one of the main bottlenecks in packet processing [28] and (ii) the LLC is the target of packet transfers to/from the NIC (DDIO). Due to space concerns, the above numbers are presented only when they are needed (i.e., when dropping traffic).

All the code used for testing is publicly available.\textsuperscript{5}

4 DROPPING TRAFFIC

4.1 Pure I/O performance

Results in Figure 2 show the performance in terms of dropped traffic of the technologies under test in a pure I/O scenario (packets are only touched by swapping their MAC addresses). In general, XDP packet dropping is highly efficient, as it avoids additional kernel processing (if the Linux network stack is traversed) or to exchange frames on AF_XDP rings (if AF_XDP processing is involved). However, XDP dropping performance are even better if AF_XDP sockets are enabled, with a 21% improvement when the driver is executed in interrupt mode (XDP-sk). Numbers in Figure 3, which shows LLC accesses (hits and misses), seem to suggest a higher memory usage of pure XDP, which needs one LLC load and one LLC store per packet, while other technologies do not need the store operation. However, this LLC store is due to a preflchw() operation in the NIC driver, which prepares the memory area to store the metadata for packet transmission (the xdp_frame operation). However, since the preflchw() assembly instruction is executed asynchronously, we detected no difference in performance when removing the above operation from the driver, even when this memory region is not needed (i.e., when dropping traffic).

We speculate that the root cause of the performance improvement in XDP-sk is the different buffer management model introduced by AF_XDP, since, to the best of our knowledge, this is the only main difference when enabling AF_XDP sockets in the drop scenario. In addition, Figure 4 shows the execution context of the packet processing code, namely user space, system call or software interrupt, where the last two are both in kernel space. Interestingly, the usage of a system call to retrieve packets is convenient when dropping traffic in user space (AF_XDP sync), but it has a negative effect when this operation is performed in the kernel (XDP-sk sync), since we still have to spend some processing time in user space just to trigger another driver loop. Finally, Figure 4 shows also that technologies that include a context switching (e.g., part of the processing is done in user space, part in SoftIRQ) tend to perform worse, suggesting the opportunity to choose a technology that completes all the processing in the same context.

In conclusion, dropping packets at the kernel level always proved to be more efficient (XDP, XDP-sk and XDP-sk sync), with an extra improvement when enabling AF_XDP sockets (XDP-sk), suggesting a more effective buffer management model introduced by AF_XDP.

\textsuperscript{4}https://github.com/FedeParola/memoslap
\textsuperscript{5}https://github.com/FedeParola/xsknf
4.2 Impact of memory demand

Figure 5 shows the throughput achieved by the most effective I/O configurations (namely XDP, AF_XDP-sysc, and XDP-sk) while increasing the memory allocated to our NF and randomly accessed. Results show a stable throughput as long as the allocated memory is limited in size, with an advantage of in-kernel (XDP and XDP-sk) over user space (AF_XDP). However, the gap shrinks when the amount of allocated memory exceeds 16K cache lines (corresponding to the size of our L2 cache), with user space processing eventually outperforming XDP first and XDP-sk next.

Comparing these results with the number of per-packet accesses in the LLC cache (Figure 6) we can see that they remain stable as long as the size of the allocated memory fits in the L1/L2 caches. All LLC accesses in this region are related to a load/store in the packet buffer; in fact, DDIO transfers (through DMA) packets to/from the LLC, hence all packet accesses result in an LLC access. As in the pure I/O test case, pure XDP presents one additional LLC access due to a prefetch operation. While this operation should intuitively affect the memory scalability, we did not detect any performance difference when removing that operation. In general, pure XDP performance showed to be the most affected by memory demand, with an improvement achieved when enabling AF_XDP sockets (XDP-sk). However, user space processing with AF_XDP (AF_XDP-sysc) proved to be the most resilient solution with respect to memory demand, which suggests the presence of some important difference beyond the buffer management model, whose identification is left for future studies. For instance, this is confirmed in Figure 8 (e.g., 5M entries), in which AF_XDP-sysc is less efficient when a limited memory is involved, but it outperforms XDP-sk when a large amount of memory is requested (in that case, the identifiers of many TCP sessions).

In general, processing packets in user space with AF_XDP proved to be less affected by the amount of memory accessed by the NF.

4.3 Impact of CPU demand

Figure 7 shows the throughput achieved by the technologies under test with NFs of increasing processing complexity. Unlike the memory case, all technologies were similarly affected by this increase in CPU requirements, with in-kernel processing keeping its performance gap over user space (even if this became less relevant with higher processing complexity becoming dominant over the I/O processing cost). As expected, we did not detect variations in the number of LLC/memory accesses, that always remained equal to the ones measured in the pure I/O test. Interestingly in this test we did not experience the gap between the XDP and the XDP-sk modes, likely because the NF processing cost dominates over the cost of pure I/O.

4.4 Traditional NF performance

In this experiment we evaluated the performance of a realistic NF, namely a simple L4 firewall dropping all traffic whose 5-tuple matches a set of pre-configured rules stored in a hash table. The main parameter that influences its performance is the number of ACL entries, as well as the number of different flows matching those entries. Therefore, in our test we scaled this variable, influencing the degree of memory dependency of the function.

We wrote two versions of the firewall, an XDP-based and an AF_XDP-based one, keeping them as similar as possible. To avoid a bias in the results due to the characteristics of the hash table, the AF_XDP firewall leveraged a user space clone of the eBPF hash map available in kernel, which differs only in how concurrent read and

---

Footnote: For the sake of precision, the data structure has the following features: it uses the list_nulls double linked list of the kernel to store elements that collide on the same bucket; the maximum size of the map is defined at initialization time and the number of buckets is defined rounding this number to the next power of two; the memory for map nodes (storing both the key and the value) is pre-allocated in a contiguous area; the hash function is used for hashing and the hash value is mapped to the corresponding bucket selecting its lower n bits (with 2^n buckets); when an element is modified (added, deleted or updated), the concurrent access to the corresponding bucket is protected with a spin lock.
write operations are handled. The eBPF hash map leverages ReadCopy-Update [17], which supports multiple reads to be executed in parallel with one update, with a negligible cost on the read side. Unfortunately there is no ready-to-use implementation of this mechanism in user space, so we decided not to support this type of concurrency in our map and our tests included only read operations, in which the overhead would be negligible anyway; hence, we pre-populated the maps before starting our measurements.

We configured the firewall with an increasing number of ACL entries (each one with a different source IP address), and tested the maximum throughput achievable by the NF when dropping all the received packets, which were randomly distributed across all configured sessions. Results in Figure 8 confirm the advantage of dropping packets at XDP level (both with AF_XDP sockets enabled and with standalone XDP) but only as long as the size of the ACL is limited (and fits the L2 cache size). XDP reaches up to 29% higher throughput compared to AF_XDP sysc, but this advantage tends to shrink as the number of sessions increases (i.e., when the cost of memory access tends to dominate over pure I/O cost). When the size of the ACL exceeds 10K entries, there is no appreciable difference between the two technologies, and for an even larger number of sessions, dropping packets in user space becomes much more efficient with a 51% performance improvement over XDP. This confirms the better memory efficiency of AF_XDP found in our Memory demand test: while a hash map makes hard to predict in advance when the data structure starts generating accesses in the LLC, our cache measurements (omitted for the sake of brevity) show an increase of LLC accesses in the 10K-100K sessions range.

Takeway 1: Dropping packets at the XDP level (XDP and XDP-sk) is more efficient when the packet processing function operates mainly in the lower levels of cache (L1 and L2), thanks to the smaller amount of code traversed to process the traffic. Bringing packets in user space (AF_XDP sysc) results advantageous when the NF becomes more memory bound, hence representing the most effective solution for memory intensive processing logic.

5 PASS-THROUGH TRAFFIC

5.1 Pure I/O performance

We evaluated raw packet I/O performance by measuring the maximum throughput when performing a swap of MAC addresses and sending back the packet out of the receiving interface (return code XDP_TX at the XDP level).7 Figure 9 shows that the clear winner is AF_XDP sysc, with a 29% improvement over XDP. With respect to user space processing, AF_XDP sysc outperforms AF_XDP, hence differing from the dropping test case in which interrupt-based or system call-based modes brought to similar performance. Instead, AF_XDP yielded a lower throughput, which can be explained with the need to periodically perform a system call to inform the driver of the presence of new packets to transmit. This represents an operation that consumes a CPU time similar to the (less efficient) poll() based mode, as shown in Figure 11 (AF_XDP and AF_XDP poll bars). Curiously, when redirecting packets with XDP-sk*, we always obtained a very low throughput (about 2.5 Mpps). The reason may be

This paper does not include the results when the traffic is redirected on a different interface, which can be achieved with return code XDP_REDIRECT at the XDP level, and with minor changes at the AF_XDP level. In the above conditions, our experiments showed lower performance for both XDP and AF_XDP, with a trend that is very similar to the presented one.
due to the number of LLC accesses shown in Figure 10: while the number of LLC loads per packet is similar for all tests (and in line with the results of the drop test case), a notable difference exists for store operations. This number ranges from 0.1 per packet (with AF_XDP) to one per packet (with XDP) (unlike the drop test case, here the additional LLC write is needed to populate the xdp_fram metadata for transmission; see Section 4.1) till two per packet (with XDP-sk). In fact, an analysis of the Linux kernel (at least till version 5.14) shows that when AF_XDP sockets are enabled and packets are re-transmitted at the XDP level, the packet is copied from its UMEM frame to an in-kernel XDP page before being sent out of the interface. This expensive operation is probably the cause of the higher number of LLC stores and consequent lower throughput of the XDP-sk test cases.

5.2 Impact of memory demand

Results of the memory test shown in Figure 12 and Figure 13 underline a trend similar to the one observed in the dropping traffic scenario, with AF_XDP broadening its gap over XDP as memory accesses shift from L1/L2 caches to the LLC and to main memory. This gap goes from a 19% improvement when the function uses little memory to a maximum of 39% higher throughput when a notable number of accesses hits the LLC and the main memory.

5.3 Impact of CPU demand

Similar considerations apply to the CPU-intensive test (Figure 14): XDP and AF_XDP sync score similar, with the limited advantage of the latter that disappears when increasing the complexity of the processed function, i.e., when the raw I/O cost becomes less relevant. The reduced memory efficiency of XDP-sk has a huge impact in this CPU-intensive test as well, hence sitting at the bottom.

5.4 Traditional NF Performance

To assess the performance of traffic redirection with a realistic function, we wrote a minimal load balancer that parses the packet till level 4 and uses the 5-tuple to access an hash map of the active sessions. If the lookup is successful, the retrieved value contains the information to update the packet, that can be either the destination IP, port and MAC address of the backend for incoming packets (client to service), or the original service IP and port for return packets (service to client). In case the lookup fails, a new load balancing decision is taken and two entries are added to the table of active sessions to handle both incoming and return packets. In the end, the fields of the packet are updated and the packet is sent back on the receiving interface. As presented in Section 4.4 with respect to the user space implementation of the hash map, we limited our performance measurements on packet processing scenarios involving only lookup operations, avoiding write operations. Therefore, we populated in advance the table of active TCP sessions of the load balancer generating traffic that is randomly distributed among those sessions.

Figure 15 shows the maximum throughput handled by the NF for an increasing number of active sessions. For a small number of sessions (i.e., limited memory used) the performance advantage of AF_XDP against XDP is reduced compared to the one measured with raw I/O performance (a 14% improvement vs the 29% recorded in the former test). However, when the number of sessions increases...
and the NF becomes more memory bound, the same behavior observed in the dropping scenario applies, with user space processing increasing its performance gap over XDP, with a maximum of 38% higher throughput when processing packets of 100K different sessions.

**Takeaway 2**: AF_XDP sysc, i.e., AF_XDP sockets with system call-triggered driver, provides the highest performance when processing pass-through traffic. The advantage over XDP is limited for simple packet processing functions but increases as the logic becomes more memory bound, requiring frequent accesses to the LLC and to the main memory. In general, considering the higher flexibility of user space processing that, unlike eBPF, does not impose any limitation (e.g., on the program size, on the type of data structures used, on loops, etc.), AF_XDP sockets should be preferred for pass-through traffic with respect to in-kernel processing.

## 6 LOCAL TRAFFIC

This section investigates the case of traffic processed by one or more NFs before landing on an application running on the local server, usually as a container (e.g., a Kubernetes pod). In this case, the traffic has to traverse the entire TCP/IP stack before data is eventually delivered to the application. In our analysis we do not consider the case of applications running in a virtual machine because of the increasingly diffusion of cloud-native workloads, particularly with respect non-NF applications running in telco-oriented datacenters.

While processing a packet at XDP level, the XDP_PASS return code can be used to inform the kernel that the packet has to continue its journey in the standard network stack and reach the application running locally. Instead, for what concerns AF_XDP, traffic is handled in user-space and therefore we need a way to re-inject packets into the kernel, which is needed to complete the remaining TCP/IP processing and deliver the packet to the application.\(^9\) Therefore we leveraged a veth interface to re-inject the packet in the kernel; however, this introduces more overhead due (i) to the necessity to perform an additional copy of the packet, as veth devices do not support the zero-copy mode,\(^9\) and (ii) to the additional context switch. Therefore, given that the zero-copy capability is a property of the UMEM, we relied on two different UMEMs, the first one bound to the physical interface and operating in zero-copy, while the second one associated to the veth and operating in copy mode. This solution requires an additional copy of the packet between the two UMEMs each time a packet traverses the two interfaces, resulting in two copies per packet, the first in user space (with AF_XDP) and the second in the kernel (on the veth). Unfortunately, leveraging a single UMEM shared among the two interfaces does not solve the problem. In fact, while the first (user-space) copy would be avoided, this method requires the physical interface to work in copy mode, resulting anyway in two copies per packet for local traffic (performed by the kernel on each one of the two interfaces), but losing the performance advantage of zero-copy on the physical interface in case of pass-through traffic.

\(^8\)We did not encompass user space implementations of the TCP/IP stack (e.g., [12]) since they are not widespread and require patched applications in order to be leveraged.

\(^9\)At the time of writing, all physical and virtual NICs supporting XDP are also compatible with AF_XDP sockets, but only physical NICs support the more efficient zero-copy mode, even if an attempt to add it to the veth driver was made in the past [25].

---

![Figure 16: Path of traffic reaching a local application, when the NF runs as XDP code (blue continuous line) or as AF_XDP code (red dashed line).](image1.png)

![Figure 17: Throughput when delivering a packet to the TCP/IP stack based on the different processing paths highlighted in Figure 16.](image2.png)

While with AF_XDP the traffic is processed on the ingress and the egress paths (thanks to the packet redirection in user-space done by the veth), XDP programs can only be attached to ingress hooks. In this case, egress traffic processing leverages the TC eBPF hook, which executes the same packet processing function on traffic leaving the server. Figure 16 highlights the different paths followed by the traffic to reach a local application, depending on where the NF is implemented (XDP or AF_XDP).

### 6.1 Pure I/O performance

The first test determines the raw I/O performance limited to the early part of the Linux TCP/IP stack, by assessing the overhead of adding custom NF processing to packets reaching the local application. Particularly, it assesses the cost of the initial processing of the packet before reaching the TCP/IP stack, which is different in XDP and AF_XDP + veth cases, while the remaining processing stack is the same for both. We used Moongen to generate UDP traffic with non-existing MAC addresses, and executed an XDP/AF_XDP program that performs a MAC swap on the packets, which are then passed to the network stack. Due to the wrong destination MAC address, which does not correspond to the ones present on the server, packets are dropped very early in the network stack. Results in Figure 17 show that adding some processing at the XDP level has very little overhead (9%) with respect to the baseline (i.e., when packets are dropped by the kernel without being first processed by XDP/AF_XDP). On the other hand, when moving packets into user
space and then back into the kernel, the hit on performance is considerable, with a 64% reduction of throughput. Interestingly we did not experience any difference for the three different AF_XDP-based modes.

On the other side, XDP-sk* experiments measure the cost of enabling AF_XDP sockets on traffic that is not redirected in user space. This is the price we may have to pay if we want to leverage both XDP and AF_XDP at the same time, e.g., by splitting the traffic and handling it in part with AF_XDP (which represents the best choice for pass-through traffic, Section 5), in part with XDP (which represents the best choice for dropped traffic, Section 4). One of the reasons of this additional cost is due to the necessity to copy the received packet from the UMEM to a new buffer before sending it up to the network stack, assuming the NIC operates in the more efficient zero-copy mode. This is needed because the UMEM can be modified in user space, hence the packet could be corrupted during kernel processing, causing unexpected behaviors. Vice versa, in vanilla XDP the buffer is only accessible by the kernel, hence allowing true zero-copy operations [15]. This cost could be prevented by using AF_XDP sockets in copy mode that, like XDP, receives packets in a private kernel buffer and then copies them to the UMEM. This however would result in very poor performance for the traffic reaching the user space (in our experiments we were able to achieve a maximum of 1.17 Mpps when redirecting packets in copy mode, way below the performance of other zero-copy versions of AF_XDP shown in Figure 9), hence forcing us to discard this alternative. Results show that, at least in this I/O-intensive test, the overhead of enabling AF_XDP sockets is significant, with a 22% performance reduction of XDP-sk against pure XDP (Figure 17).

The second test analyzes the raw I/O performance including the entire Linux TCP/IP stack and the receiving application, by using our sample application (memcached), hence evaluating the impact of added packet processing in a real scenario. In this test, the number of CPU cores to be used, and the allocation of different processing tasks to each core proved to be more problematic than in the previous cases. For instance, the Linux scheduler gets confused by the polling-based working mode of AF_XDP, which looks like a user-space process always requiring more resources. Hence, the CPU allocation is partitioned among all the requesting processes (in this case AF_XDP and memcached), reaching the best equilibrium when the core is equally shared among the two (50% each). However, this would achieve sub-optimal performance, because the 50% allocated to AF_XDP is only partly spent in doing actual processing, while the rest is spent in empty busy polling iterations. To overcome the above limitation, in this test we used multiple cores. We dedicated one core to packet processing and then we added a number of cores to memcached that enabled to reach stable performance, which implies a saturation the NF core. This lead to the usage of (1+3) cores, achieving an average 96% usage of memcached cores, indicating that we were wasting almost no resources. It is important to notice that with this configuration the ingress TCP/IP stack is executed on the NF core (where the software interrupt of the veth interface is triggered), while the egress stack is executed on the application cores (where the system call sending packets is executed). In fact, when we leveraged Receive Packet Steering (Section 2.3) to move the network stack processing to the memcached cores, we simply increased load imbalance (i.e., some fully utilized cores along others with more idle time), resulting in lower overall throughput, hence confirming that the previous configuration was the best choice in our operating conditions.

For the XDP-sk sync experiments, even if packets are processed at the XDP level, the code is executed in the context of the user space application, within a busy loop triggering the driver with a system call. From the Linux scheduler perspective, the NF is therefore seen as an application requiring all resources available on the core, and is subject to the same considerations that apply to AF_XDP, hence we handled it with the same (1+3) cores partitioning. For what concerns other solutions based on in-kernel, interrupt-based processing (i.e., XDP and XDP-sk), we were able to partition available cores in a more granular way thanks to RSS, allowing the NF and memcached to share each one of the 4 available cores according to their needs and the overall traffic load. The final cores distribution used in our test is shown in Figure 18.

With XDP, we achieved the highest performance by enabling Application Targeting Routing (Section 2.3), that moves the XDP processing on the same core where the recipient application is running (Figure 19, XDP ATR on). This technology is not available when processing traffic in user space because it requires the execution of a part of the driver that is bypassed by AF_XDP. Since not all network cards support ATR (or equivalent technologies), and it might not always be effective (e.g., in the case of connectionless traffic), we evaluated the performance of in-kernel processing also with ATR off (i.e., relying on the classic RSS packet steering). Figure 19 shows the throughput we achieved in terms of requests per second handled by memcached. The gap between in-kernel and user space packet processing reduced significantly with respect to the former test case (Section 6.1), but running our NF at the XDP level still guaranteed a considerable lead over AF_XDP, with a throughput advantage of 42% with ATR on, and 27% when this technology is not available. In this test, the performance reduction when enabling AF_XDP sockets (i.e., XDP-sk) is limited to no more than 5%.
Figure 20: TCP/IP-based application throughput (memcached) when applying some complex processing (load balancing) to packets.

Final remark, no NIC-based accelerations were used with XDP-sk sysc and all AF_XDP-based technologies, as the entire traffic goes on the single core that executes the NF. Instead, XDP and XDP-sk look more efficient when ATR is on, compared to the simpler RSS (active when ATR is off). In the XDP context, enabling RFS did not prove to have any positive impact with respect to plain RSS. This is due to the fact that RFS is applied at a later stage in the networking stack with respect to XDP, after the NIC driver has completed its operations, resulting in the XDP program and the application thread potentially being executed on different cores.

6.2 Traditional NF performance
To evaluate the impact of a more complex NF, we extended the load balancer presented in Section 5.4 to balance sessions also toward a local memcached backend running on the same server. Results in Figure 20 are very similar to the ones that we observed in the simple processing scenario (Figure 19), indicating that the cost of the NF is negligible compared to the complexity of the network stack and the application. However, our tests show that in the AF_XDP* and XDP-sk* cases the additional pressure put on the (single) NF core exacerbates the load imbalance problem caused by the rigid partitioning of cores. In fact, in this test the bottleneck imposed by the NF core caused our application cores to be leveraged only at about 88% (compared to the 96% of the mac swap test), slightly increasing the gap between in-kernel and user space processing performance.

Tentative guideline 3: When processing traffic that is directed to a service running on the local server and leveraging the TCP/IP stack, XDP is much more efficient than AF_XDP, thanks to its smooth integration with all the TCP/IP processing code that runs also in the kernel, hence avoiding expensive kernel-to-user (and vice versa) context switches. Moreover, XDP facilitates distributing the processing power of the CPU cores in a simpler and more granular way between NFs and applications, particularly when ATR is available.

7 DISCUSSION AND SUGGESTED BEST PRACTICES
In previous sections we characterised the performance of the analyzed packet processing technologies on homogeneous classes of traffic (either dropped, redirected out of the machine or delivered to a local application). While for each one of the above categories we were able to identify the most efficient processing technology for the NFs, a one-size-fits-all winner does not exist. Hence, this section leverages previous results to provide some guidance for real world deployments, focusing on resource-limited edge datacenters in which each server may need to handle all the three types of traffic (dropped, forwarded, terminated locally) at the same time, with the highest efficiency. To do so, we derive two preliminary guidelines that could drive the design and optimal placement of NFs, starting from the simple combination of the takeaways presented in the previous sections, which assumed the presence of a single class of traffic. Then, in the following sections we verify whether these guidelines hold also in real world scenarios, with a combination of traffic of different classes.

- **Tentative guideline 1.** When handling only pass-through and dropped traffic, process it in user space with the system call-triggered driver (AF_XDP sysc), thanks to its higher performance (which is even more evident when a huge amount of memory is requested, due to its higher efficiency in that case) and superior processing freedom (no eBPF limitations) (Tentative 2). In addition, offload only the most accessed packet dropping rules in the kernel (as long as they fit in the L2/L3 cache), in order to leverage earlier packet discarding without incurring in the memory penalty of XDP (Takeway 1), while the rest is left to user-space.

- **Tentative guideline 2.** When handling also local traffic, process this class of traffic in the kernel, to avoid the expensive crossing of the user-kernel barrier multiple times (Takeway 3). However, in case a NF needs to operate on all types of traffic, this may require to duplicate its logic both in user space and XDP, which might not always be possible due to the limitations of eBPF. In this case, the developer could evaluate whether (other) existing kernel networking facilities (such as qdisc, netfilter, etc.) can be used to achieve the desired function, overcoming the limitations of eBPF. Alternatively, he can move local traffic to user space, incurring in the additional cost of re-injecting packets into the kernel to send them to the application through the TCP/IP stack, unless he can modify general-purpose applications to receive the TCP/IP traffic directly from user-space.

7.1 Mixing pass-through and dropped traffic
To evaluate the effectiveness of Tentative guideline 1 we defined a NF chain composed of our firewall followed by the load balancer. We compared the performance of the chain running (i) purely in user-space (i.e., AF_XDP), (ii) purely in kernel-space (i.e., XDP), and (iii) in hybrid mode, in which the firewall logic is at the XDP level and the load balancing logic sits in user space. In this first test, which does not include local traffic, we generated a total of 11K flows: 1K matched the ACL of the firewall (hence were dropped), while the remaining 10K reached the load balancer. Figure 21 shows the throughput globally handled by the chain (both dropped and redirected packets) varying the share of traffic belonging to each class. Curiously, the performance advantage of user-level processing with respect to XDP when all packets are redirected decreased from 33% when only the load balancer

\footnote{Our proof-of-concept implementation repeats the parsing of the packet both in kernel and user space. A possible improvement would be leveraging XDP metadata to share information already computed in the eBPF program with the user space [2], whose evaluation is left as a future work.}
was used (Figure 15) to 19% in this case, suggesting that the user
space code seems to be more affected by the additional ACL lookup
than XDP. Unfortunately moving the firewall logic in the kernel
(\textit{Hybrid sysc} in Figure 21) highly impacted the performance of the
chain also in the scenario where all traffic was redirected (20% throughput reduction with respect to \textit{AF\_XDP sysc}), making even
pure XDP processing more effective than the hybrid approach. The
performance advantage of both XDP and the Hybrid approach over
full user space is noticeable only when the share of dropped traffic
exceeds 75%. While in this scenario, with huge amount of dropped
traffic, XDP performs slightly better than the Hybrid solution, the
latter is much more suitable to be enabled dynamically, hence
allowing to switch from pure AF\_XDP to Hybrid upon necessity.
In fact, this requires only to replace the existing XDP program
with a new one (an atomic operation without service disruption),
while switching from AF\_XDP to XDP requires creating/destroying
sockets and/or changing the interrupt configuration of the NIC
queues, which can hardly be done with the current technology.

\textit{Takeaway 4}: When handling only pass-through and dropped
traffic, process all traffic in user space under normal conditions (i.e.,
when most traffic is forwarded) and dynamically offload the most
accessed packet dropping rules in the kernel (as long as they fit in
the L2/L3 cache) when the amount of dropped traffic is predominant,
for example during the mitigation of a DDoS attack.

### 7.2 Mixing pass-through and local traffic

The \textit{Tentative guideline 2} speculates that the best choice for pass-
through traffic is AF\_XDP in user space, while local traffic stays in
kernel with XDP. However, to have both the above technologies
running at the same time, we need some processing logic that ana-
lyzes incoming traffic and redirects each packet to the appropriate
pipeline. In this section we assume to have both pass-through and
local traffic and we evaluate the feasibility and effectiveness of this
hybrid approach, analyzing the different options for the processing
logic that separates the two classes of traffic.

First, we analyze a software-based solution running at the XDP
level, which redirects part of the traffic in user space for AF\_XDP
processing, while the rest continues along the TCP/IP stack (\textit{XDP\_PASS}
return code). Then, we will explore an hardware-based alternative,
which leverages the capabilities of the NIC to steer different flows
to different receive queues. These tests used a load balancer NF,
which redirects local traffic to the proper pod replica of the final
application (running on the server itself), while the pass-through
traffic was redirected to multiple external servers. In case of traffic
splitting, the load balancing logic of each instance will operate only
on the portion of traffic handled on that path (e.g., either local or
pass-through).

#### 7.2.1 Software-based splitting

In this paper we assume that the traffic terminating locally can be detected by simply checking the
5-tuple of the packet, which enables to detect traffic directed to
local destinations with a simple lookup on a hash table. To assess
the feasibility of this solution we started by measuring the overhead
of the additional XDP dispatching logic on the traffic redirecting
performance of the \textit{AF\_XDP sysc} load balancer, in the optimal case
in which the hash table of local sessions contains only one element
that is never hit (we generated only pass-through traffic). Results
in Figure 22 show that the overhead of this solution (\textit{AF\_XDP sysc + XDP disp}) greatly affects overall performance, compromising the
benefits of user space processing (\textit{AF\_XDP sysc}) and making even
a pure XDP implementation of the load balancer more appealing
in case part of the traffic reaches local applications through the
TCP/IP stack. This result does not surprise since the dispatching
logic has almost the same complexity as the load balancing program,
and it is not needed in case of pure XDP. Since the addition of the
required splitting logic reverted previous results and made XDP a
better solution even in the pass-through-only scenario, which is the
preferred battlefield for AF\_XDP, we avoided additional tests with
local/dropped traffic that, as confirmed by our previous tests, are
already pushing further for an XDP-based solution. Nonetheless, a
pure software-based splitting mechanism may still be useful, for
example when eBPF limitations prevent the implementation of the
correct logic at the XDP level, and therefore we need to rely on
deeper kernel network stack facilities (e.g. the TC layer, Netfilter).
In this case the performance benefit of AF\_XDP sockets over the
kernel stack for pass-through traffic may overweight the cost of
flows dispatching.

\textit{Takeaway 5}: When handling also local traffic (in addition to
pass-through/dropped one), and no hardware-based packet steer-
ing mechanism is available, process all the traffic in the kernel
with XDP, given the prohibitive overhead of software-based packet
dispatching.

#### 7.2.2 Hardware-based splitting

An alternative to software (XDP-based) flows dispatching is to rely on the capabilities of the NIC
to steer different flows to different receive queues, leveraging the

![Figure 21: Pass-through + dropped traffic: effect of kernel offloading of packet dropping logic (Hybrid sysc) if compared to pure user space (AF\_XDP sysc) and pure in-kernel (XDP) processing.](image)

![Figure 22: Pass-through traffic only: impact of additional in-kernel packet dispatching logic applied to AF\_XDP sysc (AF\_XDP sysc + XDP disp) compared to pure XDP or pure AF\_XDP sysc.](image)
AF_XDP mode (with system-call-triggered driver) and another set (due to the dedicated CPU cores), until we need to reallocate a core from the set dedicated to local traffic to the one dedicated to pass-through traffic, at the expense of an inferior power gradually being re-allocated from local traffic / application to pass-through traffic, leaving RSS/ATR dispatching mechanisms available, (ii) a more complex and rigid partitioning of resources (e.g., CPU cores) is acceptable and (iii) a high volume of pass-through traffic is handled, split pass-through and local traffic between user space and in-kernel processing to achieve best overall performance.

For the sake of precision, the above experiment assumes that the traffic splitting operation is simple, e.g., based on IP destination addresses or the session 5-tuple. In some cases, e.g., edge clusters running 5G mobile core components, the traffic processing may include some heavy operations (e.g., User-Plane Function - UPF, GTP de-tunneling) in order to derive the above parameters, raising the question where to implement the above (expensive) processing. The analysis of the above case is left for future work.

8 RELATED WORK

Different works in the recent literature studied the properties and characterized the performance of in-kernel and user space packet processing with a focus on the XDP and AF_XDP technologies.

In [10] Hohlfeld et al. analyze the performance of offloading packet processing both to the kernel and to a SmartNIC leveraging XDP. While advantages and drawbacks of these two approaches are thoroughly studied, the paper does not analyze the possibilities of interaction between in-kernel and user space processing. In [27] authors propose an architecture for eBPF based NF's called eVNF. This architecture encompasses a fast path executed at the XDP level to carry out simple but critical tasks, while a slow path, based either on the kernel network stack or on a user space application accessible through AF_XDP sockets, handles corner cases. Both these papers
however, probably due to the early stage of development of the AF_XDP technology, do not consider user space packet processing as a valid fast path alternative.

In [24] authors present the maintainability, flexibility and performance considerations that led to the selection of AF_XDP as the base for the future data plane of Open vSwitch. The paper identifies the performance limitations of user space when processing traffic directed to containers leveraging the TCP/IP stack, and provides some preliminary considerations on the possibility to rely on in-kernel processing for this type of traffic. Our work extends these considerations. [1] proposes a different solution to the problem, experimenting with a user space implementation of the TCP/IP stack to support standard applications and layer 7 NFs.

[22] makes the case for automatically decomposing eBPF/XDP based NFs in multiple programs and between in-kernel and user space components to bypass the limitations imposed by the eBPF verifier.

8.1 DPDK user space drivers

The most common solution for high-speed packet processing is currently DPDK coupled with custom user space drivers, which enable direct access to the hardware from user space. While the DPDK can leverage AF_XDP and standard Linux drivers for packet I/O, user space drivers present multiple advantages with respect to AF_XDP, at the cost of taking complete control over the NIC, which is no longer visible by the kernel. In fact, an all-user space design enables higher performance because it avoids expensive system calls and/or context switches. Furthermore, unlike AF_XDP programs, user space drivers can access many hardware offloadings features such as TSO and GRO, as well as packet metadata (e.g., checksum) provided by modern NICs.

However, we did not consider the DPDK technology in our paper because it is more appropriate for dedicated servers, which may not be the case of small edge-based data centers as well as cloud-native deployments, as suggested also by [24]. DPDK requires the use of memory hugepages, and the mmap()ing of large contiguous memory areas, an operation that could fail. Both these constraints complicate the coexistence with other traditional applications. Moreover, user space drivers would prevent native traffic processing in the Linux TCP/IP stack, given the full control of DPDK over the NIC, requiring to re-inject packets from user space into the kernel, whose (low) performance is expected to be similar to the one showed in Figure 17. On the other hand, AF_XDP can easily enable part of the traffic to be natively processed by the Linux kernel TCP/IP stack, with traffic steering done either in software (with an XDP program) or with rules installed on the NIC.

A possible way to enable a similar behavior with DPDK would be leveraging SR-IOV with user space drivers bound to one or more Virtual Functions (VFs), the kernel bound to the Physical one (PF), plus the proper additional logic at the NIC level to steer packets among the two as in Section 7.2. This logic however might not be supported by the NIC, as in the case of our Intel XL710, where each VF must have a different MAC address used for packet steering, that would make VFs and the PF behave like completely different interfaces.

Given the above considerations, it is worth noting that, in a context in which no traffic has to be processed locally and packets are either dropped or forwarded, DPDK’s user space drivers can represent a more efficient choice with respect to the results presented in Section 4 and Section 5, which builds on the assumption of having shared server. We leave a holistic evaluation, encompassing all available technologies, as a future work.

9 CONCLUSIONS

This paper presents the performance characterization of in-kernel and user space packet processing based only on the recent kernel-provided XDP/AF_XDP infrastructure, without the need to maintain and integrate custom kernel modules. Our analysis focuses on the scenario of telco edge data centers, where the processed traffic includes both a pass-through portion, handled by a chain of NFs and redirected towards a remote destination, as well as a local portion, directed to applications running on the same set of servers and leveraging the local TCP/IP stack.

We carried out a set of experiments studying these classes of traffic, with the aim of optimizing the usage of (scarce) edge resources by running both data plane-oriented workloads and traditional applications on the same (shared) servers. Our results underline which technology is best suited for each possible mix of traffic, deriving six guidelines that can help telecom operators to select the best technology based on the actual operating conditions, and to achieve optimal performance in a mixed workload scenario such as the one present in the data centers at the edge of the network.

ACKNOWLEDGMENTS

Federico Parola acknowledges the support from TIM S.p.A. through the PhD scholarship. This work was partially supported by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on “Telecommunications of the Future” (PE00000001 - program “RESTART”).

REFERENCES


An attempt to support the above features in AF_XDP was proposed in [3].
A REPRODUCTION OF OUR RESULTS

To encourage the reproduction of our results, the code used for our tests is available online at https://github.com/FedeParola/xsknf. In this appendix we discuss our experiments in terms of Repeatability (i.e., ability of our team to obtain the same results multiple times), Reproducibility (i.e., ability of independent teams to obtain the same results using their own artifacts) and as described in the ACM Artifact Review and Badging document.\(^\text{12}\)

A.1 Repeatability

Given the controlled environment in which our experiments were carried out, with few to none external sources of variability, our results are highly repeatable.

A.2 Reproducibility

Anyone can expect to reproduce the same results presented in this paper with the setup presented in Section 3.3 and the code published in the online repository. The repository contains a library to speedup the development of NFs based on XDP and AF_XDP and to simplify their integration. Under the examples folder it is possible to find all the NFs used in our experiments. Each one is composed, at the bare minimum, by an in-kernel XDP program (`<nf>_kern.c` file), a user space program, encompassing the management and configuration logic as well as the user space AF_XDP data plane logic (`<nf>_user.c` file), and a common header file (`<nf>_h.h`). The tests folder contains detailed information on how to reproduce our tests. Tests based on connectionless (UDP) traffic can be automatically executed through the python scripts contained in the folder. Each script needs to be configured with the parameters of the testbed (e.g., IP addresses of the DUT and tester server) as well as the parameters of the test (e.g., sizes of the firewall ACL). The scripts leverage the `pktgen` tool.

In the following table we provide additional information on software versions, omitted from Section 3.3 for the sake of brevity.

<table>
<thead>
<tr>
<th>Software</th>
<th>Version</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>9.3.0</td>
<td>User NFs compilation</td>
</tr>
<tr>
<td>Clang/LLVM</td>
<td>10.0.0</td>
<td>eBPF NFs compilation</td>
</tr>
<tr>
<td>libxdp</td>
<td>1.2.3</td>
<td>Embedded in the repo</td>
</tr>
<tr>
<td>memcached</td>
<td>1.5.22</td>
<td></td>
</tr>
<tr>
<td>MoonGen</td>
<td>Commit 525db91</td>
<td></td>
</tr>
<tr>
<td>memoslap</td>
<td>Commit f9dbf</td>
<td></td>
</tr>
</tbody>
</table>

A.3 Replicability

In our experiments we paid particular attention to code and data structures in order to make the implementation of our user space and XDP NFs as close as possible, with the aim to focus on the performance of the I/O technologies and not on the NFs themselves. Anyone trying to repeat our measurements with different code should therefore attain to the same principle.

Beyond the code of the NFs under test, other factors that may affect results are the hardware and the software environment. With respect to the latter, since XDP/AF_XDP is a new technology under quick evolution, the version of the kernel as well as the driver of the NIC may heavily influence the results.

---

\(^{12}\)https://www.acm.org/publications/policies/artifact-review-and-badging-current