Ultra-low Latency NFV Services Using DPDK

Leonardo Lai*, Gabriele Ara*, Tommaso Cucinotta*, Koteswararao Kondepu†, and Luca Valcarenghi*

**Scuola Superiore Sant'Anna*, Pisa, Italy †*Indian Institute of Technology Dharwad*, Dharwad, India
{name.surname}@santannapisa.it k.kondepu@iitdh.ac.in

***Abstract*—This paper introduces UDPDK, a novel middleware easing the implementation of ultra-low-latency communications among software components, based on the well-known DPDK framework, by which one can bypass the in-kernel networking stack of the operating system, with direct access to the networking devices from the application. DPDK functionality is exposed to applications with an API resembling the standard POSIX primitives for UDP. The usefulness of UDPDK is demonstrated by integrating it in the OpenAirInterface open-source software stack for RAN communications, modifying its F1-U component to use UDPDK primitives. Experimental results on an LTE test-bed show a reduction of 69% of the end-to-end latency when using the new primitives. The achieved results are of foremost importance for the realization of ultra-low-latency communications in future 5G scenarios.**

***Index Terms*—DPDK, Radio Access Network (RAN), Network Function Virtualization (NFV), OpenAirInterface, Low Latency**

I. Introduction

Interconnecting billions of mobile devices globally in a fast, affordable, accessible, flexible, and scalable way represents one of the most ambitious challenges of nowadays and future networks. These need to support high-performance networking use-cases requiring high bandwidth and low latency, ensuring high mobility and coverage, and low power consumption.

To this end, the 5th Generation (5G) architecture aims at enabling three broad classes of usage scenarios: enhanced Mobile Broadband (eMBB), massive Machine-Type Communications (mMTC), and Ultra-Reliable and Low-Latency Communications (URLLC) [1], [2]. eMBB supports data-intensive services that require stable high-bandwidth connections, such as needed in mobile AR/VR applications, and UltraHD or 360-degree video streaming; URLLC supports services with ultra-low latency and very high reliability requirements, as needed in safety-critical systems; finally, mMTC aims at supporting connectivity of a significantly higher number of devices than we have nowadays, as needed in IoT networks with unprecedented device density and low-power requirements.

These new networking scenarios are pushing towards higher dynamicity and flexibility in resource management, with increased decoupling between hardware and software through abstraction layers heavily based on *virtualization*, drawing from lessons learned in the area of Cloud Computing. These requirements, coupled with the convergence of networking technologies over TCP/IP, led to the widespread adoption of Network Function Virtualization (NFV) and Software-Defined Networking (SDN) architectures: instead of having dedicated physical networking appliances sized for the peak hour, we have software-based network functions, i.e., Virtual Network Functions (VNFs), that can be deployed as elastic scalable software components over a NFV infrastructure [3], [4]. This is essentially a general-purpose private cloud infrastructure of the network operator that is able to flexibly instantiate, consolidate, relocate, upgrade and elastically scale VNFs according to time-varying demand conditions.

Sample network functions that can easily be implemented in software and deployed as VNFs include DHCP, NAT, routing, switching, proxying, deep packet inspection, and radio [5]. Indeed, over the past few years, research in the field of efficiently applying cloud computing and virtualization paradigms to NFV and SDN infrastructures has flourished [6].

Due to the higher levels of resource sharing promoted by NFV, proper countermeasures are needed to avoid excessive interferences among them to ensure high and predictable performance of the deployed VNFs. Therefore, NFV defines the concept of *network slicing* to facilitate the coexistence of heterogeneous services enabling dynamic allocation of resources to them while guaranteeing isolation and pre-defined levels of Quality of Service [7]. When VNFs are deployed on general-purpose cloud servers, temporal isolation among co-located functions can be achieved by system partitioning and core pinning [8] or using real-time scheduling disciplines [9], [10] when a finer-grained resource allocation is needed. This makes it easy to deploy services in NFV infrastructures and scale them dynamically upon necessity, being particularly useful for those applications characterized by many users but low-bandwidth demand for individual streams, like in IoT.

Among others, NFV has been recently applied to base station functions (namely gNB) even when they are split into different units, such as Radio Unit (RU), Distributed Unit (DU), and Central Unit (CU) [11], as depicted in Fig. 1. Different functional split options present different requirements in terms of capacity and latency; thus care shall be taken to select the best virtualization [12] and networking technology.

In this context, it is crucial to ensure fast and efficient communications among the various services, both when deployed in geographically distant servers and when co-located. Technologies to accelerate virtualization exist both in hardware and software. Intel VT, Data Direct I/O Technology, and Single Root I/O Virtualization (SR-IOV) help increase the bandwidth while reducing the latency on the hardware side. Software solutions (e.g., DPDK [13] and netmap [14]) allow optimizing the network stack to these specific use-cases, bypassing the typically feature-rich – thus bulky – networking stacks realized

This work has been partially supported by the EU Commission through the 5GROWTH project (grant agreement no. 856709).

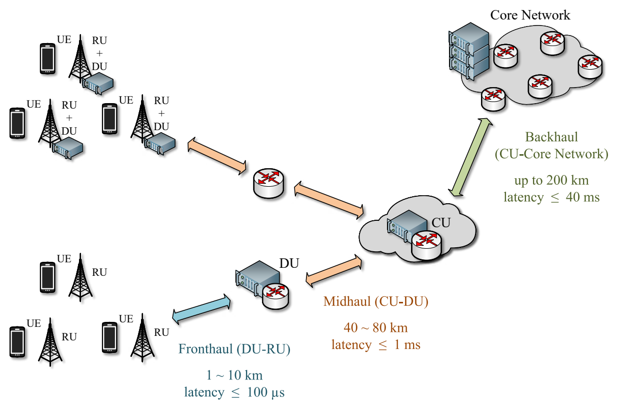


Figure 1: Sample 5G Radio Access Network (RAN) architecture per O-RAN/3GPP specification, with the annotation of the typical latency constraints among communicating components, and their implications in terms of maximum physical distance.

in an operating system (OS) kernel, dropping unneeded features. Instead, these realize only the strictly needed functionality directly in user-space, exploiting technologies for direct access to networking devices, resulting in higher performance levels. These frameworks usually leverage techniques like continuous polling, I/O batching, zero-copying, and others.

A. Contributions

This paper investigates the application of high-performance networking frameworks in the 5G software stack. Specifically, our study focuses on using Data Plane Development Kit (DPDK) to accelerate communications in the Central Unit/Distributed Unit (CU/DU) functional split. With reference to Fig. 1, our purpose is to optimize communications on 5G Midhaul links (i.e., the F1-U interface) between CU and DU components, so to achieve the lowest possible end-to-end latency. This is beneficial considering the critical latency requirements imposed by various 5G use-cases.

To this end, we present a novel middleware that we developed to ease the adoption of DPDK technologies from higher-layer software stacks, focused on replicating a set of functions that resemble the typical system calls available on an OS for UDP-based communications. Then, a set of modifications to the OpenAirInterface (OAI) open-source software are described, which, leveraging on the introduced middleware, is now capable of realizing ultra-low latency communications for user plane exchanges within the F1-U component of the OAI/5G/vRAN architecture, with the traffic being vehicled through DPDK. An experimental evaluation of the obtained software stack is performed on a real test-bed, where we measured a 69 % reduction of the end-to-end latency compared to using traditional socket-based communications.

II. Related Work

Several works addressed how to optimize the performance of networking primitives in the research literature and industrial practice, and particularly trying to leverage technologies like DPDK. In the following, we review the most important DPDK-based middleware solutions; then, we review works aimed specifically at applying DPDK in the context of NFV and Virtualized Radio Access Network (vRAN).

A. DPDK Frameworks

Various frameworks provide high-performance network stacks, often on top of high-performance low-level frameworks like DPDK [13] or netmap [15], and sometimes relying on code "borrowed" from widespread operating systems like Linux or FreeBSD. mTCP [16] is a user-level TCP stack designed for scalability on multi-core systems; it is developed on top of the PacketShader engine for efficient event-driven packet I/O [17], using DPDK for low-level operations and device accesses. F-Stack [18] is a user-space TCP stack built on DPDK with a POSIX-like application-programming interface (API), which facilitates the conversion of existing applications to the framework. The framework imposes an event-based programming model based on the `epoll` and `kevent` system calls, which may represent a limitation when attempting to port applications designed differently. DPDK Accelerated Network Stack [19] is a DPDK-based network stack that implements complete TCP/UDP and IP protocol stacks, including ARP, ICMP, DHCP, and other protocols.

The FD.io Transport Layer Development Kit (TLDK) [20] provides a set of libraries for high-performance implementations of TCP and UDP protocols. It is maintained to support the development of Vector Packet Processing (VPP) [21], a high-performance software switch. Since it is highly specialized to meet VPP requirements, its API is not meant to be compatible with POSIX sockets, limiting its applicability.

SeaStar [22] is an event-driven framework that provides its own TCP/IP stack, optionally using DPDK to manage network devices in user-space. IPAugenblick [23] is a porting of the Linux TCP/IP stack to DPDK that relies on a background process acting as a glue component between poll-managed devices and applications. Its API is POSIX-alike, but the project does not see any active development since 2016.

LibOS Network Stack in Userspace (LibOS-NUSE) [24] is a subsystem for Library Operating System (LibOS) that provides a network stack entirely implemented in user-space. LibOS is POSIX-compatible and can also be used with applications relying on standard system calls for networking. LibOS-NUSE also supports high-performance frameworks like DPDK, but it lacks support for batch packet processing, which constitutes a significant performance limitation [25].

To summarize, most of the mentioned frameworks are difficult to use to accelerate software 5G frameworks in NFV with UDP-based interactions among components due to several limitations: (**i**) discontinued development support (IPAugenblick); (**ii**) the lack of support to POSIX-like API (SeaStar, IPAugenblick, TLDK), which would greatly facilitate porting existing software; (**iii**) the lack of UDP as transport-level protocol, which forces applications to use TCP, with its implied overheads (mTCP, SeaStar); (**iv**) forcing applications to adopt specific programming patterns, often event-driven (mTCP, F-Stack, SeaStar); (**v**) being purposely developed for custom operating systems (LibOS-NUSE).

As a motivational and driving example, the F1 User Plane Protocol (F1-U) interface implementation in OpenAirInterface (OAI) is implemented on top of the POSIX UDP socket API provided by Linux. The most straightforward way to accelerate this connection is to use a high-performance middleware that provides POSIX-like APIs, provides (minimal) UDP/IP support, and does not impose a specific application pattern or forces the usage of specific API calls (like `kevent/epoll`). This is exactly what we propose in this paper.

B. DPDK Adoption in vRAN, NFV, and SDN

vRAN architectures possess critical end-to-end latency and jitter requirements that pose severe limitations on what virtualization technologies and CU/DU functional split options are viable, as shown in [12]. Therefore, various attempts exist at accelerating packet processing in this domain. In [26], Field Programmable Gate Array (FPGA)-based acceleration is combined with the OpenCL framework to accelerate some physical layer functions of the 5G stack, outperforming CPU-only processing for large wireless channel bandwidth. Similar conclusions are also drawn in [27], where authors show that an FPGA-accelerated regular-expression matching engine in NFV has a similar performance to a DPDK-based accelerated one, but it scales much better when operations are parallelizable.

Moreover, various works exist in the research literature trying to use DPDK for achieving enhanced performance in such contexts as vRAN, NFV, and SDN. For example, the challenges in realizing a cloud-distributed MU-MIMO RAN system are presented in [28], where DPDK is leveraged for accelerating communications over 10 GbE links, and Intel Streaming SIMD Extensions are leveraged to accelerate encode/decode operations. Also, a DPDK-based NFV architecture is presented in [29], where monolithic VNFs are disaggregated into lightweight "micro" VNFs, enabling a finer-grained resource allocation and reducing redundancy in the deployed stack. In [30], a high-performance software switch for vRAN is presented called N-VDS. It reuses principles similar to DPDK, but on top of the VMWare vSphere Distributed Switch architecture. This achieves a 3x to 5x throughput gain compared to other virtual switches, including OVSDPDK.

Some works also exist in the context of accelerating SDN with DPDK. For example, it has been shown [31] that in OpenFlow software switches DPDK acceleration may compensate for the typical performance drop generally due to the interactions with the OpenFlow controller. Focusing on security-related NFV functions, in [32] authors present results obtained accelerating NFV functions related to deep packet inspection using DPDK primitives coupled with SR-IOV. The results show that the DPDK-based implementation outperforms the one based on the `libpcap` framework of the Linux kernel, allowing for much higher throughput (roughly 8x).

Finally, albeit not belonging to the NFV context, it is interesting to see what impact DPDK may have in other ultra-low latency applications, such as realizing game servers, as done in [33]. The presented experimental results show a latency reduction of nearly an order of magnitude with the adoption of DPDK, albeit authors observe that having to reimplement the networking stack in user-space is quite cumbersome for developers. This reinforces the message that the availability of open-source frameworks easing the adoption of DPDK via a POSIX-alike socket API is a good thing to have, as provided by our proposed framework, described next.

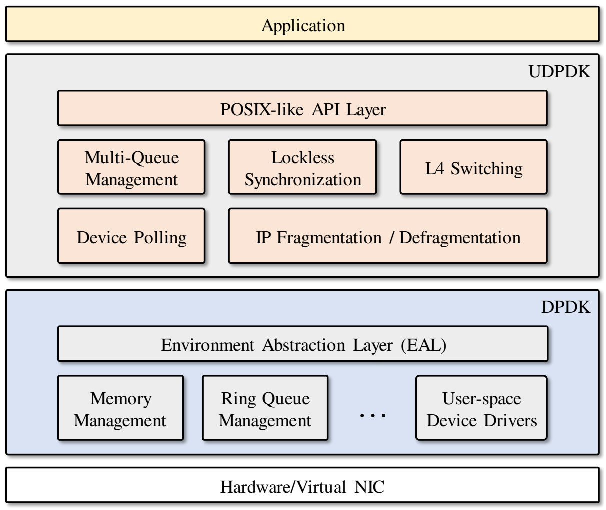


Figure 2: UDPDK application stack.

III. Proposed Approach

This section presents a novel middleware that we developed on top of DPDK to streamline the adoption of DPDK into applications originally developed using other networking paradigms, like the POSIX networking API. We describe the main features of the software library, and we summarize its internal implementation. We then show the changes that we made to the OpenAirInterface (OAI) F1-U components as a successful use-case for the proposed middleware.

A. UDPDK

UDPDK is a DPDK-based middleware that we developed to address critical shortcomings of existing solutions to high-performance networking. Typically, applications are either developed using a standard POSIX socket API, leveraging in-kernel networking stacks, or are directly implemented on top of high-performance networking frameworks (e.g., DPDK, netmap), each with their own specialized APIs and programming paradigms.

UDPDK provides a minimal user-space UDP-IP network stack designed to be both efficient and easy to integrate with existing and novel applications. It is implemented on top of DPDK Environment Abstraction Layer, but at the same time it does not force its users to a programming paradigm completely different from POSIX socket APIs. In fact, the API exposed by UDPDK is purposely designed to resemble as much as possible traditional sockets (albeit with limited functionality), reducing the effort required by application developers to port existing applications to DPDK. UDPDK is released under an open-source license at https://github.com/leoll2/udpdk.

POSIX socket API calls implemented by UDPDK include: `socket`, `bind`, `getsockopt/setsockopt`, `sendto/recvfrom`, and `close`. These functions, conveniently defined in UDPDK using the `udpdk_` prefix to avoid confusion with actual POSIX sockets, tightly emulate the behavior of their POSIX counterparts and, as such, they exhibit similar usage. Not every socket option is supported, and at this moment UDPDK does not support key functions like `select`, `connect`, `send/recv`, and the general `sendmsg/recvmsg`, as well as their batch counterparts `sendmmsg/recvmmsg`, which will be the subject of future updates to the framework. For now, UDPDK supports only the `AF_INET` domain and the `SOCK_DGRAM` protocol (a.k.a. UDP/IP network stacks), and not all functions of the network and transport-level protocols are implemented yet. Nevertheless, UDPDK can already be effectively used in fairly common practical NFV use-cases, as shown later. UDPDK does not aim to replace the native socket interface completely; instead, the two APIs may coexist so the programmer can choose which underlying stack (kernel or DPDK) to use. In particular, we expect UDPDK to be used to accelerate data plane communications, while communications on the control plane (typically less data-intensive and more sporadic) are performed through OS sockets.

Fig. 2 depicts the stack of an application developed on top of UDPDK. The main components of UDPDK (also shown in Fig. 2) are the following: the *Poller* handles all interactions with DPDK devices, accessed in a polling fashion; the *Multi-Queue Manager*, implemented on top of DPDK's lockless ring queues (`rte_ring`), manages a configurable number of efficient packet queues, two per opened UDPDK socket (namely a *reception* and a *transmission* queue per socket); incoming/outgoing packets for each UDPDK socket are exchanged using these queues; the *L4 Switch* multiplexes incoming packets to the corresponding socket, based on the destination UDP port number of each packet; the *IP Fragmentation/Defragmentation Manager* either fragments outgoing packets or reassembles incoming packets before delivering them to the *L4 Switch*; this component leverages existing DPDK libraries that provide methods for splitting, buffering, and rebuilding packets based on IP/UDP header fields.

Notice that UDPDK does not implement some functions typically performed by L3 network stacks, most notably, UDPDK lacks a proper implementation of ARP and ICMP protocols[1]. The UDPDK target use-case is the one of point-to-point communications within an L2 domain, where most of the traffic can be easily sent to recipients without caring about routing and other aspects relevant for networks with non-trivial topologies. Nevertheless, generated IP packets still retain valid headers (even if not all options are supported) and can be exchanged over local networks without issues.

[1]For now, a table of ARP associations is provided to UDPDK during its initialization phase.

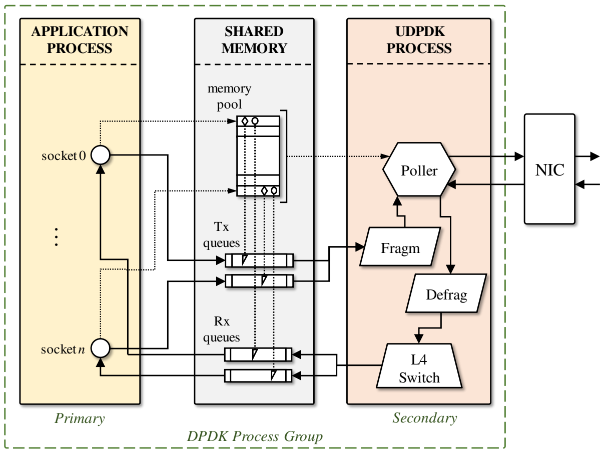


Figure 3: Interactions among UDPDK internal components.

Interactions between these components are shown in Fig. 3. In general, UDPDK applications are multi-process applications, with one dedicated polling process (spawned by the middleware framework upon initialization) and one or more application processes. More precisely, on its start-up, UDPDK handles the initialization of the underlying DPDK layer, including all virtual/physical NIC devices and memory pool configuration. It then initializes a *DPDK Process Group*[2] and spawns a *DPDK secondary process*, later referred to as the *UDPDK Process*. This process will handle all communications with the underlying devices and perform all operations typically implemented in a (minimal) networking stack. The main application process is then assigned the role of the *DPDK primary process*[3].

Once all processes are up and running, application processes can open one or more UDPDK sockets. When opened, each socket is assigned a pair of incoming/outgoing message queues. From that point on, the socket can be used to send or receive data. In the former case, all data provided by the user through `udpdk_sendto` is copied to a memory area from the reserved memory pool, and then the allocated buffer is enqueued on the outgoing queue. The *UDPDK Process* continuously polls outgoing queues of all sockets, constructs outgoing packets (filling L2/L3/L4 headers information), performs IP fragmentation (if needed), and then proceeds to send all constructed packets to the underlying networking device.

Once this operation is concluded, the same process proceeds to poll the NIC for new incoming packets, collecting all available packets, performing IP defragmentation, if necessary. It then delivers the content of each packet to the

[2]A *DPDK Process Group* identifies all processes that share DPDK-managed resources, including opened NIC devices, memory, and data structures.

[3]Running application processes as *DPDK secondary processes* may not be possible, due to some technical problems. In particular, DPDK does not deliver any interrupts coming from network devices to secondary processes, and this could impair the application logic being executed. In general, this does not represent a limitation to applications with a single application process or applications that use a dedicated process for high-performance network communications; in both of these cases, relevant processes will run as *DPDK primary processes* when using UDPDK, avoiding these limitations introduced by the underlying DPDK layer.

corresponding destination queue, dropping packets that cannot be delivered properly. Once new packets are present in the reception queue, the application process can collect them by calling `udpdk_recvfrom`. All packet creation, processing, and management operations are efficiently implemented on top of the high-performance API provided by DPDK, using shared-memory buffers, forcing cache-line alignment of all data structures (to minimize cache misses), and packets are processed in batch when possible. The same shared memory area is used to store both packets data and descriptors, as they are accessed and manipulated by both application processes and the *UDPDK Process*.

Finally, UDPDK provides a packet generator (*pktgen*) and a *ping-pong* application to demonstrate its capabilities.

B. Accelerating OpenAirInterface with UDPDK

To demonstrate the applicability of UDPDK in high-performance NFV and vRAN scenarios, we modified a subset of OAI components to use UDPDK sockets instead of POSIX ones. Our modifications to OAI are released in open-source and made available at: https://github.com/leoll2/oai_dpdk.

Data exchanged by the CU and DU components of the OAI F1-U interface is encapsulated in Protocol Buffers, with one task for each component handling all data encapsulation and exchange over the local network. These two tasks (called *agents* in OAI terminology) are located in `F1AP/f1ap_cu_task.c` and `F1AP/f1ap_du_task.c`, respectively; they initialize the communication channels and spawn a thread each to exchange messages. More in detail, the `proto_agent_start()` initializes a channel structure, registering the send, receive and release callbacks, allocating a POSIX socket, and finally invoking `create_link_manager`. This last function manages all transport-level protocols used by OAI (TCP, UDP, and Stream Control Transmission Protocol), and it spawns two threads with real-time scheduling priority (`SCHED_RR`) that periodically attempt to transmit/receive data, respectively, using FIFO queues to exchange data with other components of the program.

Since the F1-U interface implemented by OAI already uses UDP as transport-level protocol to exchange data over the network, accelerating communications between OAI CU and DU components is relatively easy with UDPDK without disrupting much of the original application flow. After including the initialization and cleanup of the UDPDK framework within appropriate points of the OAI initialization and cleanup code, most of the substitutions necessary for this port simply require changing calls to `sendto/recvfrom` to corresponding UDPDK calls by adding the corresponding UDPDK API calls by adding the signature prefix `udpdk_`, as well as substituting all sockets creation and configuration calls to UDPDK ones. The relevant code is located in the `openairinterface5/openair2` directory, which groups all transport-level protocols supported by OAI.

In particular, the UDP socket is created within the `new_link_udp_server` function, which relies on the `socket` primitive, which we substituted with `udpdk_socket`. The functions `link_send_packet/link_receive_packet` call `sendto/recvfrom` respectively, which we changed to `udpdk_sendto/udpdk_recvfrom`. Finally, UDPDK initialization and teardown calls are inserted within the `USER/lte-softmodem.c` file, where all other high-level OAI functions are called. Proper placement of the `udpdk_init`, `udpdk_cleanup`, and the `udpdk_interrupt` signal handler[4] complete the set of changes to the OAI code needed to port its F1-U CU/DU components to UDPDK. Of course, the OAI build process (based on CMake) needs to be slightly modified to include the UDPDK library as an optional dependency, where a boolean option can be used to enable/disable UDPDK-based acceleration.

IV. Experimental Results

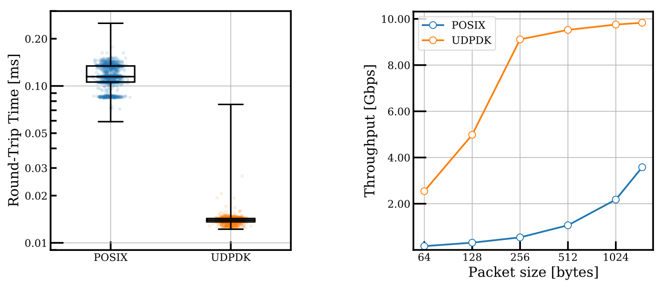
To evaluate the performance of the proposed framework and the benefits of the OAI acceleration that we implemented on top of it, we used a small proof-of-concept testbed simulating a 5G Non-Stand Alone network composed of a DU, a CU, and a Core Network (CN), hosted on three separate servers on the same local network. We deployed the DU and CU components on two identical Dell PowerEdge R630 V4 servers, each equipped with two Intel Xeon E5-2640 v4 CPUs clocked at 2.40 GHz, 64GB of RAM, and two directly connected Intel X710 DA2 10 Gbps Ethernet Controllers. The CN component was instead deployed on a low-end UP-Board, featuring an Intel Atom CPU clocked at 1.44 GHz, 4 GB of RAM, and a generic off-the-shelf network card. Both the CU and DU components used a Linux kernel version 4.15.0, while the CN used on a slightly older 4.7.7 kernel.

The CU and the DU were mutually connected with two distinct interface pairs to separate the control plane (F1-C) from the user plane (F1-U). The F1-U traffic was forwarded through the DPDK-enabled 10 Gbps network interface, whereas the F1-C used a standard 1 Gbps interface. The CN machine hosted the Mobility Management Entity, Home Subscriber Service, and Serving Gateway/PDN Gateway entities together. In this experimental setup, the CU and the CN were connected through a VxLAN to form the backhaul network. The Remote Radio Unit was exploited with the help of the LimeSDR[5] open-source software-defined radio, and it was attached to the DU through a USB interface. Note that the LimeSDR was used for completeness to have a correctly operating deployment of all three components. This way, we could measure the F1-U latency between the DU and CU, which in the end was utterly unaffected by the CN (inferior) hardware.

In all experiments, we disabled Intel Turbo-Boost and manually fixed the CPU frequency to 2.40 GHz. Experiments with UDPDK use DPDK version 20.05 and the DPDK implementation of the Virtual Function I/O driver, while when using the POSIX socket API the device driver was `i40e`.

[4]A signal handler is not necessary for UDPDK to function, but it ensures proper cleanup is performed when the application is closed upon reception of an external signal.

[5]More information at: https://limemicro.com/products/boards/limesdr/.



(a) (b)

Figure 4: Performance comparison between POSIX sockets and UDPDK when using a 10 Gbps cable. (a) shows the **RTT** between two equivalent pairs of *ping-pong* applications exchanging 64 byte packets; whiskers show maximum and minimum values obtained over more than 15 thousand samples per socket type. The Y axis uses a logarithmic scale. Lower is better. (b) shows the mean **throughput** measured between two equivalent pairs of *sender-receiver* applications for increasing packet sizes up to 1500 bytes per packet. The X axis uses a logarithmic scale. Higher is better.

Note that in the text below "packet size" refers to the size of the whole L2 frame, and throughput values are computed considering the size of the whole L2 frames exchanged.

Before running the complete testbed with OAI, we performed some experiments using the UDPDK *ping-pong* and *sender-receiver* application pairs to demonstrate the remarkable difference between using standard sockets and UDPDK. Fig. 4 shows results achieved deploying application pairs on our two directly connected hosts, the same that will later be used to run the CU and DU components during tests with OAI. Each pair (*ping-pong* and *sender-receiver*) has been first tested using POSIX sockets and then using UDPDK. Both latency and throughput tests (Fig. 4a and Fig. 4b, respectively) show clearly that UDPDK outperforms POSIX, achieving on average about one-tenth of the round-trip time (RTT) of the corresponding POSIX-based applications (a 88 % RTT reduction) and up to about 17 times the throughput (a 1580 % throughput improvement).

Then, we moved on to test the effectiveness of the proposed modifications to OAI. As described before, the OAI F1-U implementation uses an asynchronous interface with two threads and FIFO queues to send and receive messages. Our experiments measure the round-trip time between the CU and DU components, either when data is exchanged asynchronously through message queues (`message_get`/`message_put`) or when packets are synchronously exchanged using the "low-level" OAI API (`link_send_packet`/`link_receive_packet`). We performed experiments in three scenarios: using UDPDK, using POSIX sockets with UDP as transport layer, or using POSIX sockets with TCP. In all scenarios, the asynchronous API is expected to achieve lower round-trip times than their synchronous counterparts since the `message`-based API calls

Table I: OAI F1-U interface: RTT as a function of the transport-layer stack used.

Protocol	Stack	Queue	Average RTT [ms]
UDP	UDPDK	No	0.029
		Yes	0.031
UDP	POSIX (Kernel)	No	0.095
		Yes	0.125
TCP	POSIX (Kernel)	No	0.275
		Yes	0.370

internally invoke the `link` ones. Table I shows the results obtained with all test combinations; the reported round-trip time values represent average values of more than a thousand samples collected over ten repeated runs of each experiment configuration. In general, UDPDK exhibits a much lower latency than the in-kernel UDP stack used through the POSIX API (e.g., a 69 % reduction from 95 µs down to 29 µs), thanks to its ability to bypass the kernel entirely, which in turn outperforms the in-kernel TCP stack as available through the POSIX API, as expected.

Moreover, UDPDK seems to couple reasonably well with the asynchronous interface, with only a minimal extra overhead introduced by the message queue between the OAI application threads. As expected, the TCP runs turn out to be the slowest due to the greater complexity of the involved software stack and protocol. These results confirm the validity of the proposed approach since UDPDK effectively reduces the latency between two critical components of the data plane.

Finally, we must add that these performance improvements come at the cost of spawning a dedicated process (the *UDPDK Process*) that continuously polls network devices and incoming queues from each open UDPDK socket. Moreover, when applications need blocking calls from the standard socket API, the new UDPDK calls do not block the calling process, but they wait for results to become available, performing an active busy-wait. This behavior is typical of DPDK-based solutions, and it generally results in at least two or more CPU cores constantly reaching 100 % utilization [34], [35], regardless of the actual ongoing traffic. DPDK can also be used in combination with interrupts [36], but before sending or receiving packets the program must switch back to polling mode. This reduces CPU utilization during idle times, at the cost of a more significant latency when interrupts must be disabled to revert to polling mode when the first packet of a burst is received. Alternatively, application processes can use UDPDK with the non-blocking API calls (using the `O_NONBLOCK` flag) and perform some other action while waiting for packets to be ready to be sent/received to/from the *UDPDK Process*, instead of performing continuous busy-loops on packet queues. However, in this case the cost of a single CPU fully busy due to the *UDPDK Process* itself is anyway unavoidable.

V. Conclusions and Future Work

This paper introduced UDPDK, a novel middleware for easing integration of DPDK-based communications in application-level software, thanks to its API mimicking the standard POSIX one for UDP-based communications. This is a convenient tool to accelerate VNFs in NFV infrastructures leveraging on the capabilities of DPDK. We integrated UDPDK within OpenAirInterface (OAI), an open-source software stack for packet-processing in RAN scenarios. We presented experimental results showing how the use of UDPDK reduces latency for end-to-end communications in synthetic "ping-pong" micro-benchmarks, both when performed directly with UDPDK and when integrated within OAI for communications between 5G Midhaul components.

Regarding possible future work directions, it would be interesting to complete UDPDK with additional L3/L4 functionality, which is still missing, like handling the ICMP and ARP protocols, performing IP routing across multiple NICs, and supporting commonly useful UDP flags. This would likely imply investigating the scalability of UDPDK, with the possibility to use multiple switching threads (as its current architecture uses only one thread, albeit supporting multiple sockets at the same time). Also, it would be interesting to evaluate better the impact of DPDK-based communications on real LTE/5G applications, as opposed to the synthetic benchmarks herein adopted. Finally, we would like to investigate adaptive techniques to switch between polling and non-polling modes in UDPDK to avoid the high computational cost of DPDK-based solutions compared with traditional interrupt-based networking and achieve significant power consumption reductions, retaining most of the performance improvements associated with DPDK.

REFERENCES

[1] M. Series, "Minimum requirements related to technical performance for IMT-2020 radio interface (s)," *Report*, pp. 2410–0, 2017.

[2] T. Fehrenbach, R. Datta, B. Göktepe, T. Wirth, and C. Hellge, "URLLC services in 5G low latency enhancements for LTE," in *2018 IEEE 88th Vehicular Technology Conference (VTC-Fall)*. IEEE, 2018, pp. 1–6.

[3] R. Guerzoni *et al.*, "Network functions virtualisation: an introduction, benefits, enablers, challenges and call for action, introductory white paper," in *SDN and OpenFlow World Congress*, vol. 1, 2012.

[4] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015.

[5] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications surveys & tutorials*, vol. 18, no. 1, pp. 236–262, 2015.

[6] J. DiGiglio and D. Ricci, "High performance, open standard virtualization with nfv and sdn," *Wind River*, 2013.

[7] P. Popovski, K. F. Trillingsgaard, O. Simeone, and G. Durisi, "5G wireless network slicing for eMBB, URLLC, and mMTC: A communication-theoretic view," *IEEE Access*, vol. 6, pp. 55 765–55 779, 2018.

[8] L. Cao, P. Sharma, S. Fahmy, and V. Saxena, "Nfv-vital: A framework for characterizing the performance of virtual network functions," in *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, Nov 2015, pp. 93–99.

[9] T. Cucinotta, L. Abeni, M. Marinoni, A. Balsini, and C. Vitucci, "Reducing Temporal Interference in Private Clouds through Real-Time Containers," in *2019 IEEE International Conference on Edge Computing (EDGE)*, July 2019, pp. 124–131.

[10] ——, "Virtual Network Functions as Real-Time Containers in Private Clouds," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, July 2018, pp. 916–919.

[11] ITU-T, "Transport network support of IMT-2020/5G," GSTR-TN5G, Feb. 2018, version 1.0.

[12] F. Giannone, K. Kondepu, H. Gupta, F. Civerchia, P. Castoldi, A. Antony Franklin, and L. Valcarenghi, "Impact of virtualization technologies on virtualized ran midhaul latency budget: A quantitative experimental evaluation," *IEEE Communications Letters*, vol. 23, no. 4, 2019.

[13] "DPDK," https://www.dpdk.org/, [Online] Accessed September 5, 2019.

[14] L. Rizzo, "Revisiting network I/O APIs: The Netmap framework," *Queue*, vol. 10, no. 1, p. 30, Jan. 2012.

[15] ——, "Netmap: a novel framework for fast packet i/o," in *21st USENIX Security Symposium (USENIX Security)*, 2012, pp. 101–112.

[16] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mtcp: a highly scalable user-level TCP stack for multicore systems," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 489–502.

[17] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 195–206, 2010.

[18] Tencent Cloud, "F-Stack," 2017, http://www.f-stack.org/.

[19] Ansyun, "dpdk-ans (accelerated network stack)," 2017, https://github.com/ansyun/dpdk-ans.

[20] FD.io, "Transport Layer Development Kit (TLDK)," 2016, https://wiki.fd.io/view/TLDK.

[21] "Vector Packet Processing (VPP)," https://fd.io/, [Online] Accessed September 5, 2019.

[22] ScyllaDB, "SeaStar," 2014, http://seastar.io/.

[23] V. Suraev, "ipaugenblick," 2014, https://github.com/vadimsu/ipaugenblick.

[24] "LibOS NUSE," 2014, https://github.com/libos-nuse/net-next-nuse.

[25] H. Tazaki, R. Nakamura, and Y. Sekiya, "Library operating system with mainline Linux network stack," *Proceedings of netdev*, 2015.

[26] F. Civerchia, M. Pelcat, L. Maggiani, K. Kondepu, P. Castoldi, and L. Valcarenghi, "Is opencl driven reconfigurable hardware suitable for virtualising 5g infrastructure?" *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 849–863, 2020.

[27] X. Zhang, X. Shao, G. Provelengios, N. K. Dumpala, L. Gao, and R. Tessier, "CoNFV: A Heterogeneous Platform for Scalable Network Function Virtualization," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 14, no. 1, Aug. 2020. [Online]. Available: https://doi.org/10.1145/3409113

[28] D. Wang, C. Zhang, Y. Du, J. Zhao, M. Jiang, and X. You, "Implementation of a cloud-based cell-free distributed massive mimo system," *IEEE Communications Magazine*, vol. 58, no. 8, pp. 61–67, August 2020.

[29] S. R. Chowdhury, Anthony, H. Bian, T. Bai, and R. Boutaba, "A Disaggregated Packet Processing Architecture for Network Function Virtualization," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 6, pp. 1075–1088, June 2020.

[30] D. Rajan, "Achieving high performance with virtualized data plane workloads for 5g networks," in *2019 Sixth International Conference on Software Defined Systems (SDS)*, 2019, pp. 236–241.

[31] G. Pongrácz, L. Molnár, and Z. L. Kis, "Removing roadblocks from SDN: OpenFlow software switch performance on Intel DPDK," in *2013 Second European Workshop on Software Defined Networks*. IEEE, 2013, pp. 62–67.

[32] M.-A. Kourtis, G. Xilouris, V. Riccobene, M. J. McGrath, G. Petralia, H. Koumaras, G. Gardikis, and F. Liberal, "Enhancing VNF performance by exploiting SR-IOV and DPDK packet processing acceleration," in *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE, 2015, pp. 74–78.

[33] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "A study of network stack latency for game servers," in *2014 13th Annual Workshop on Network and Systems Support for Games*. IEEE, 2014, pp. 1–6.

[34] G. Ara, T. Cucinotta, L. Abeni, and C. Vitucci, "Comparative evaluation of kernel bypass mechanisms for high-performance inter-container communications," in *Proceedings of the 10th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, INSTICC. SciTePress, 2020, pp. 44–55.

[35] G. Ara, L. Lai, T. Cucinotta, L. Abeni, and C. Vitucci, "A framework for comparative evaluation of high-performance virtualized networking mechanisms," in *Cloud Computing and Services Science*, D. Ferguson, C. Pahl, and M. Helfert, Eds. Springer International Publishing, 2021.

[36] D. Géhberger, D. Balla, M. Maliosz, and C. Simon, "Performance evaluation of low latency communication alternatives in a containerized cloud environment," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, Jul. 2018.