Towards a Flexible User-Space Architecture for High-Performance IEEE 802.11 Processing

Martin Backhaus and Markus Theil and Michael Rossberg and Guenter Schaefer

Telematics and Computer Networks Research Group
Technische Universität Ilmenau, Germany
[martin.backhaus, markus.theil, michael.rossberg, guenter.schaefer][at]tu-ilmenau.de

***Abstract*—Exploiting bandwidth potentials and managing complexity of current and future IEEE 802.11 networking standards becomes increasingly difficult when using today's operating system kernels and high-throughput wireless network interfaces due to overhead caused by interrupts and kernels of complex general-purpose operating systems. These problems can be tackled by employing special purpose forwarding planes deeply integrated with wireless drivers. To do so we propose a user-space implementation of drivers and Wi-Fi stack, built upon the Data Plane Development Kit (DPDK) from the wired world. This allows for scalable frame processing, as well as flexible and easy experimenting with new protocols and approaches (compared to kernel development). Our prototypic evaluation reveals that user-space processing of IEEE 802.11 frames can increase throughput and decrease delay of a wireless connection significantly, e.g., 100 % more throughput for small frames and 27 % less delay. We also point out common obstacles when adapting wireless drivers from kernel to user space and provide advice on how to overcome them. The code base is made publicly available.**

***Index Terms*—Efficient Packet I/O, IEEE 802.11 User-Space Frame Processing, DPDK**

I. Introduction

The continuously growing need for higher data rates in communication networks has constantly been driving the development of scalable hard- and software solutions. Wireless networking standards, like IEEE 802.11ac, already provide high data rates and there is more growth to be expected in the future. The increasing importance of technologies like high-speed wireless 5th Generation Mobile Networks (5G) backhaul [1] or the Internet of Things (IoT) [2] confirms this assumption.

Furthermore, complexity of IEEE wireless standards constantly increases, which in turn leads to more complex implementations of kernel components and wireless drivers. Thus, low-level experimenting with wireless protocols becomes more difficult and there is room for improvement – in particular for special purpose services that do not necessarily use all features implemented in the kernel, e.g., a tailored forwarding dedicated to Wireless Mesh Networks (WMNs). Furthermore, the kernel causes overhead by its interrupt-based scheduling paradigm.

Both problems, the challenge of scalable throughput and managing complexity of current standards, can be solved with user-space implementations for wireless drivers and 802.11 components. In case of wired interfaces, the research community proved multiple times that full utilization of high-throughput interfaces is possible when using user-space packet I/O techniques and frameworks [3], [4], [5]. Common real-world use cases are software routers [6], firewalls, carrier-grade Network Address Translation (NAT), and load balancers [7] etc. Therefore, user-space implementations for wireless communication might be a way to handle complexity of current wireless standards and to fully utilize their bandwidth potential for more specialized use cases. The latter include, for example, research testbeds with exact timing requirements for event time stamping and delay estimation. Another example targets cost-efficient interconnection of many small IoT devices using IEEE 802.11 technology, which is likely to result in a large number of mainly small packets of measurement data or commands. Access Points (APs) or Mesh Points (MPs) could benefit from the performance of user-space approaches in this case.

Other areas of research can benefit from user-space 802.11 frame processing as well. For example in Software-Defined Networking (SDN), overall data path performance of virtual routers improved thanks to user-space processing for wired interfaces [8]. These benefits will expand to wireless SDN [9], if user-space implementations support wireless drivers as well. When wireless components converge with wired SDNs [10], user-space implementations of wireless network stacks are a necessity to achieve high performance, as packets are forwarded between wired and wireless interfaces.

Having a small, efficient architecture for Wi-Fi frame processing and wireless interface handling yields further benefits. Lower layer mechanisms are more accessible for experiments and research, e.g., targeted at cross-layer optimization. Additionally to robustness gains (since the final software will be an isolated user-space

This work was supported by RWTÜV-Stiftung (Essen, Germany) under grant number S189/10029/2016.

program, largely omitting kernel involvement), it can also be seen as an important step towards a smaller (trusted) computing base – which is desirable, e.g., with respect to security issues and formal verification. Furthermore, handling wireless hardware in user space would also allow for much simpler automated firmware testing. Further user-space benefits (besides coping with high-throughput interfaces) include: easier debugging, instrumentation and development; therefore, a faster time to market or more flexibility when experimenting.

In this paper we propose the inclusion of 802.11 processing into fast packet I/O frameworks. This incorporates running Wi-Fi interface drivers as well as a custom Wi-Fi stack entirely in user space.

In particular, we provide the following contributions:

1) An efficient architecture for a minimal Wi-Fi stack and an interface driver residing solely in user space
2) Guidelines for porting wireless drivers from kernel to user space
3) An experimental evaluation of our approach using real hardware featuring the latest IEEE 802.11ac standard comparing performance to conventional kernel processing in a high-throughput, wireless networking scenario
4) A discussion of performance bottlenecks in the Linux kernel's 802.11 subsystem in our scenario.

The remainder of this paper is organized as follows: First, we provide an overview of state-of-the-art user-space packet processing, as well as relevant research related to Wi-Fi drivers and we motivate requirements for a user-space-based Wi-Fi framework (Sec. II). We present our architecture in Sec. III and address the topic of wireless driver porting in Sec. IV. The evaluation in Sec. V discusses our approach qualitatively and studies its performance, comparing it to conventional kernel-space processing. To sum up, Sec. VI presents some concluding thoughts and directions for future research.

II. Requirements and Related Work

To include wireless interfaces into user space, an approach needs to fulfill the following functional requirements:

- Processing of frames entirely within user space (for reasons already discussed)
- The resulting implementation must conform to IEEE standards and be transparent to other wireless devices, i.e., support of multiple wireless connections and adequate configuration of the physical channel.
- It should be possible to send raw 802.11 frames or use a Wi-Fi Network Interface Controller (NIC) via a wired NIC abstraction to allow for different abstraction levels depending on the application.
- It shall provide the same level of implementation security, i.e., proper Direct Memory Access (DMA) configuration using an Input/Output Memory Management Unit (IOMMU).
- Easy and flexible instrumentation for experimenting with wireless protocols

The main quantitative requirements are:

1) high achievable throughput and
2) low latency (less or equal to current kernel implementations).

In particular, a system needs to support various wireless operation modes (Access Point, Mesh, Ad-Hoc) – including proper channel configuration. The majority of current 802.11 NICs allows the creation of virtual interfaces in hardware, so that simultaneous operation in AP and Station (STA) or Mesh mode is also valid. Support of 802.11 related security protocols, like Wi-Fi Protected Access 2 (WPA2), would greatly improve acceptance of the evolving system. In contrast to conventional kernel-mode operation, research applications require delivery of wireless management and action frames as well (configurable, to suit any need), enabling access to lower-layer features, e.g., beamforming feedback information.

The architecture should allow for a flexible usage of 802.11 frame processing, as future usage may include building high-performance access points, measuring mesh throughput or injecting forged frames for security tests.

Current NICs are available in two basic Medium Access Control (MAC) Layer Management Engine (MLME) driver models: 1) softMAC (non-timing-sensitive MLME in software, e.g., current Qualcomm chips) and 2) fullMAC (MLME mostly in hardware/firmware, e.g. current Broadcom chips). The architecture should support both. However, although softMACs are more demanding in terms of complexity, they offer access to lower-layer mechanisms and are therefore the primary target of consideration in this paper. Support for fullMAC devices could be easily added under a common configuration abstraction, like cfg80211 in Linux. Furthermore, the underlying physical device and virtual interfaces should be configurable from the user-space application. A minimal number of packet transformations should be necessary to forward frames, i.e., zero-copy I/O should be used if possible.

When looking at state-of-the-art frameworks for fast packet I/O, there are several being widely used: PF_RING [11], netmap [3] and the Data Plane Development Kit (DPDK) [12]. The implementations enable processing at wire speed – even for small frames on high-throughput network interfaces. The ideas of memory mapping (network interface buffers into memory allocated by the framework) and poll-mode drivers (instead of interrupt-driven designs) are shared among those frameworks. All of them – except for netmap – run in user-space. They differ mostly in their additional features like inter-process communication, multi-threading features and packet manipulation functions. DPDK is the current de facto standard for user-space networking, already supporting a large number of wired NICs and having an

active contributor base supported by companies like Intel, NXP, Broadcom and Mellanox. However, all frameworks only aim at processing frames from wired interfaces and hesitate to incorporate the vast complexity of IEEE 802.11 stacks. Therefore, they are not applicable for wireless communications at the moment as they lack suitable architectures to manage 802.11-related protocols.

Key differences between current wired DPDK implementations and wireless drivers in the kernel exist in the following areas:

- Data frame retransmission support in drivers
  ⇒ Necessity of efficient transmit buffer management
- Half-Duplex transmission
  ⇒ Balance reception (rx) and transmission (tx) to mitigate starvation effects
- Per-frame statistics (e.g. transmission rate, receive strength) need to be handled efficiently
- Relay some frame types to external tools, while keeping the remainder in kernel at high performance
- Complex configuration interface
- Most IEEE 802.11 NICs rely on firmware, which must be uploaded to the NICs

A common preconceived idea is the fact that poll-mode approaches are less energy efficient. This may be true for out-of-the-box examples, however some of the mentioned frameworks offer tools for means of frequency scaling and usage of interrupts in low-load periods. This leads to similar energy footprints in comparison to common kernel-based approaches. Hence, we consider energy efficiency out-of-scope for the rest of this paper.

Research in the area of device drivers provides methods for synthesizing and porting existing driver implementations. Termite [13] allows to specify a hardware interface and generate code from this specification. This is not an option, as precise specifications of current IEEE 802.11ac wireless NICs are not (publicly) available. Another tool, RevNIC [14], can be used to analyze existing (proprietary) drivers in binary form and generate a driver for a target architecture afterwards. This approach cannot generate a meaningful code structure, thus hindering further development, modification for research or even changing it from interrupt to poll mode. Finally, SUD [15] executes nearly unmodified Wi-Fi drivers in a User Mode Linux (UML) environment by means of a driver proxy in the underlying Linux kernel. Executing a Linux kernel in user space inherits its general-purpose network stack, which is not as optimized for packet forwarding performance, as applications built with user-space I/O frameworks like DPDK can be. User-space applications are also easier to extend, e.g., with software libraries.

Taking research on SDN-like approaches into account, many (e.g. [10], [16]) are evaluated using OpenWrt [17] – a Linux-based operating system for embedded devices. Although OpenWrt offers desirable functionalities and is

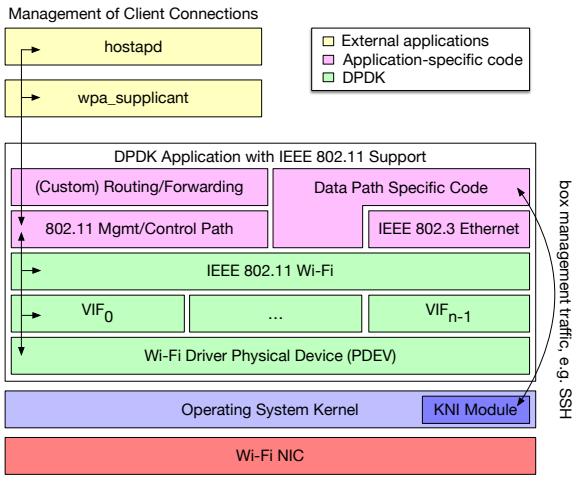


Fig. 1. Intended architecture for Wi-Fi processing in user space.

available for low-cost hardware, it inherits all drawbacks of monolithic kernel implementations as discussed above.

To the best of our knowledge, the approach presented in this paper is the first one to process Wi-Fi frames within user space using a lean and efficient architecture, which can compete against conventional kernel-space processing in terms of performance.

III. Architecture of an IEEE 802.11 Stack in User Space

This section presents an architecture for efficient 802.11 frame processing in user space. The architecture is designed in multiple layers (see Fig. 1) and integrated into DPDK. All wireless NICs supporting Peripheral Component Interconnect Express (PCIe) and DMA, may potentially be used with this architecture.

A. DPDK Background

DPDK applications create pinned threads, named logical cores (lcores), for each CPU core under DPDK control. In this article, lcore_0 denotes the first logical core, which launches all the other lcores at startup and is mostly used for performance-uncritical operations in DPDK applications. DPDK uses network interfaces exclusively. The Kernel NIC Interface (KNI), is a Linux kernel module for adding virtual interfaces between DPDK applications and the Linux kernel. It can be used to provide an interface between the Linux network stack and DPDK, in order to communicate with interfaces attached to the Linux networking subsystem if needed.

B. Architecture Overview

To allow for sustainable and efficient development, we decided to use the Linux Wi-Fi stack, especially its mac80211 subsystem for interaction with softMAC devices, as a base system. Currently, Linux is the only open source kernel with stable IEEE 802.11ac support. DPDK currently provides abstractions for dealing with IEEE-802.3-devices and useful algorithms and data structures to

build custom applications upon. Regarding Wi-Fi, such a direction also seems reasonable, which is why we followed this approach in our architecture, too.

The Physical Device (PDEV) serves configuration purposes and does not receive or transmit data directly. Furthermore, it specifies common definitions of types, constants and structs as needed in DPDK, for example rates, frame formats and IDs of frame types. A Wi-Fi PDEV exposes a `struct rte_wireless_ctx*` to upper layers, similar to the `struct rte_security_ctx*` on some IPsec or MACsec capable NICs today.

In contrast to 802.3 Ethernet, Wi-Fi has multiple frame types (data, management and control). Management and control frames may trigger long-running operations (in contrast to receiving or transmitting of one frame) and should be handled decoupled from the data plane lcores to achieve high performance and low jitter. This is done either in a dedicated thread or on the management lcore.

C. Wi-Fi Data Path

Data frames start with an 802.11 header, whereas DPDK only deals with Ethernet headers currently. Presumably, some future DPDK applications may need the flexibility to operate on raw 802.11 packets. Providing both methods can be realized by an optional abstraction for header manipulation from 802.11 to Ethernet (via an Application Programming Interface (API) or device flag).

Typical operating systems use callbacks for frame input into the 802.11 stack (mostly `ieee80211_rx_napi` in Linux, some form of `ieee80211_input` on FreeBSD). To use Linux Wi-Fi drivers in DPDK, such callbacks need to be transformed to receive functions returning arrays of packets, which boosts performance as code indirections are omitted. Efficient frame transformation from 802.3 to 802.11 is realized through mbufs (DPDK data structure for network frames and associated data) having sufficiently large headroom, to omit memory reallocations. The transmission (tx) path is most critical for high performance. A simple ring buffer of packet descriptors (`rte_mbuf`) is not sufficient to handle tx completions, as retransmissions make them non-linear. Using a bitset as free-list and an associated array of buffer pointers increases cache efficiency and speed.

D. Wi-Fi Configuration and Management Path

In contrast to the rather static configuration of Ethernet NICs, 802.11 NICs require rate configuration and other capabilities as well as optional cryptographic key material for each associated peer dynamically. We therefore use a decoupled management component for 802.11 NICs in our architecture. It continuously checks if reconfiguration of the NIC is necessary and handles new peers.

For an overview of a possible management decoupling, see Fig. 2. To avoid duplicating functionality for access point and station management, external applications such as hostapd and wpa_supplicant shall be used. This can

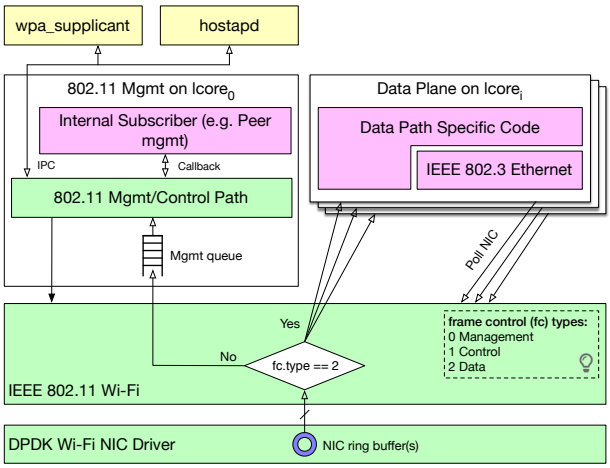


Fig. 2. Control flow of 802.11 frames.

be achieved by coupling them via Inter-Process Communication (IPC) to the management thread. Hostapd and wpa_supplicant provide a driver abstraction to interact with different configuration models. Thus, only a new interface implementation needs to be added for the DPDK implementation. Basic 802.11 management functionality also needs to be included into DPDK, as this allows for functional driver tests without external dependencies. More complex protocols like Hybrid Wireless Mesh Protocol (HWMP) should be implemented in a separate library, which can be tested independently.

During polling of wireless NICs, data and management frames can arrive. Data frames are returned to the upper layer in an array of mbufs, while management frames are enqueued in a mutex-protected queue. Enqueueing wakes the management thread, if required. Note that it is possible to add a more sophisticated scheme with multiple ring buffers and no mutex, but tests indicated no performance benefit. A DPDK-based application can either register internal callbacks for management frames or external applications like hostapd or wpa_supplicant.

IV. Adapting Wi-Fi Drivers to User Space

The majority of Wi-Fi drivers in the Linux and BSD kernels are not written in a portable way. Future porting would be much simplified, if operating system specific features would use only a small wrapper for each system called by an operating-system-agnostic base driver. Existing abstraction layers for Wi-Fi hardware of the operating systems, like mac80211, require another part in the driver specific to the operating system employed. Furthermore, high complexity of the current IEEE 802.11 standard results in large code bases of drivers and mac80211 subsystem. To evaluate our architecture with a prototype, we adapted the ath10k driver (for Qualcomm Atheros NICs) from Linux into DPDK 17.11 and adapted relevant mac80211 files. In the current 4.17 kernel the ath10k driver consists of more than 46k Lines Of Code

(LOC) and the net/mac80211 subsystem of more than 47k LOC.

Porting of Wi-Fi device drivers into DPDK in an automated fashion is not realistic, because operating systems use interrupt-based drivers with receive callbacks and often a large mac80211 interface layer. Polling-oriented drivers, as in DPDK, provide means of receiving and transmitting vectors of packets for higher performance. This requires early aggregation to packet vectors in the rx path, which conventional 802.11 drivers do not need. We use DPDK interrupt handlers on lcore_0 to obtain notifications of other events than data receive indications, which saves PCIe accesses towards the ath10k NICs. Just a single atomic counter gets incremented in the interrupt handler. This counter serves as a hint for the polling logic in the data path to check receive rings, transmission completions and management command return calls.

Incrementally porting driver subsystems of the ath10k driver layer by layer worked ideal for us, as it is a complex driver with many cross-component dependencies. Interoperability can be tested against another box operated by Linux only. For example, in case of the beaconing mechanism, the user-space implementation will be listed as a peer, if it works correctly.

If a driver directly uses `skbuff` (socket buffers, the primary Linux kernel datastructure for passing around network frames and associated data), to marshal data frames in the driver, like ath10k does, one can redefine the majority of operations on `skbuff` with functions that have similar signatures using mbuf calls from DPDK inside. Usage of `skb->data` and `skb->len` can be replaced by functions like `skb_data(skb)` or `skb_len(skb)`, for example. `skbuff` can then be a typedef of `rte_mbuf`.

Although we experimented with a first prototype, further drivers like the Intel iwlwifi wireless driver need to be ported over to DPDK. A DPDK wireless abstraction layer can only seriously be built with multiple drivers to abstract from. In our opinion, configuration of wireless cards over a `rte_wireless_ctx`, similar to `rte_security_ctx` looks most promising, as it is orthogonal to the regular ethernet device interface.

V. Evaluation

To evaluate our approach we discuss qualitative objectives first and present the experimental setup and examined scenarios, followed by quantitative benchmarking against conventional kernel packet processing. A discussion of the measured performance difference between kernel- and DPDK-based scenario concludes the evaluation.

A. Discussion of Qualitative Objectives

As desired, every component runs solely in user space. All interactions between the user-space program and the host system are reduced to Peripheral Component Interconnect (PCI) bus operations and DMA – as common in packet I/O frameworks. The IOMMU may be used to fully protect the system from faults in the network stack, but is not enabled in the prototype. Please note that in a productive environment improper IOMMU configuration via user-space code will be discarded by the kernel, therefore, user-space applications do not aggravate security concerns connected to DMA.

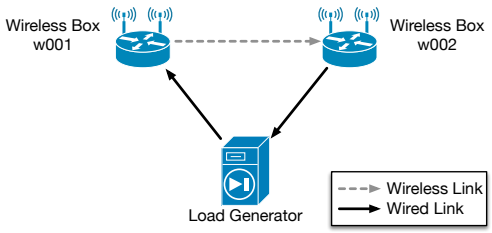


Fig. 3. Experimental setup for evaluation.

In our tests, communication between a wireless box running the user-space application and another one operated completely by Linux worked well. Thus, architectural changes are transparent and should conform to the IEEE 802.11 standard. Our implementation supports physical channel configuration and concurrent connections to multiple wireless peers. For brevity in the following quantitative evaluation, however, we restrict the experiments to a simpler setup. Concerning different wireless modes, we currently support Mesh Point operation, only. Additional modes are planned to be implemented in the future. Security-related parts of the implementation (e.g. support of WPA2) are omitted at this point, as we prioritized functional requirements that enable performance analysis for this paper. Thanks to the architecture introduced earlier in Fig. 2, we are able to handle management and action frames separately.

B. Experimental Setup

We made sure that no other wireless systems influenced the measurements, using the spectral frequency analysis of the NICs, which allows to inspect all usable Wi-Fi channels in the 5 GHz band.

Table I gives an overview of important settings and parameters of our experimental setup. Settings subject to change over the course of our evaluation (factors) are marked with "(F)". Further relevant parameters are given in subsequent text.

An illustration of our experimental setup can be seen in Fig. 3. It consists of two *Wireless Boxes* (w001 and w002) and one *Load Generator*. Every component has a separate wired management interface, which is managed by the operating system to allow SSH connections – since this is normal for an experimental testbed and we do not use the interface otherwise, we will omit it in any further discussion. Each wireless box is equipped with a Wi-Fi module based on the QCA9888 chipset (Compex WLE650V5-18, IEEE 802.11ac Wave 2, two antennas). All wireless boxes and the load generator use PC Engines APU2C4 embedded boards, equipped with three 1 GbE

TABLE I
MEASUREMENT SETTINGS AND EXPERIMENT FACTORS (F)

Setting/Factor	Used Values
Measurement runs	32 per factor combination, unless stated otherwise
Measurement duration	30 seconds
Channel bandwidth (F)	20, 40, 80 MHz
Control Channel	36 (5180 MHz)
Spatial Streams	2 (2×2 MIMO)
Guard Interval	Short
Encryption	None
Beacon Interval	1000 ms
Operating Mode	Mesh Point (802.11s)
Transport Protocol (F)	TCP, UDP
Datagram size (F)	60, 1500 Byte
UDP rate (F)	50,000 $1/s$, wire speed
Distance between nodes	4 m
Operation Mode (F)	802.11s Linux Kernel (kernel space), 802.11s DPDK-based (user space)
Linux Distribution	Debian GNU/Linux 9 (stretch)
Linux Kernel	4.14.13
DPDK	17.11
Measurement Tool TCP	iperf 2
Measurement Tool UDP	DPDK-based traffic generator

interfaces (based on Intel i210 chipset) and a quad core 1.0 GHz AMD CPU. DPDK supports i210 based NICs.

Each wireless box is connected to the load generator via 1 Gbit/s Ethernet. As Fig. 3 illustrates, traffic can be sent via Ethernet from the load generator to w001, which then relays frames over the air to w002. Eventually, all frames are being sent from w002 back to the load generator via Ethernet. A custom DPDK-based tool developed by our research group serves as load generator in our setup. The tool can be parameterized with packet size and Packets Per Second (PPS) to be sent out. It easily handles multiple interfaces and is able to saturate them completely if desired. With source and destination interfaces of the generated traffic residing on the same machine, one-way delay measurements and evaluation of test cases are easy to perform without complex time-synchronization mechanisms. Of course, well-known tools for measuring network performance (like `iperf`) can be used as well.

Clearly, there are differences concerning link load in terms of required data rate for the wireless and wired case. Due to overhead imposed by necessary wireless headers, the wireless link in the setup adds to the required rate to transmit a certain incoming wired rate.

C. Modes of Operation

Kernel mode and user-space mode target the operation of devices residing on wireless boxes, only, and are irrelevant to the load generator. In either mode of operation, we specified the same physical parameters of the wireless connection as already seen in Table I. Using smaller benchmarks, we made sure that possible performance differences between kernel mode and user-space mode are not caused by the wired interface, i.e., the wired interface does not limit performance. Specifics to the mentioned modes are explained below:

1) Kernel Mode: When in kernel mode, all interfaces (wired and wireless) are managed by the operating system. Bridging between the wired and wireless interface is active to allow for packet forwarding. The wireless interfaces are configured to run in wireless-mesh mode.

2) User-Space Mode: During user-space operation, the wireless and wired interfaces are controlled with our driver and Wi-Fi stack residing completely in user space. We developed a tool that takes care of forwarding operations in between those interfaces. For comparability to kernel mode, we ensured that our forwarding application adds the same amount of overhead in terms of 802.11 headers (also additional headers for mesh operation).

D. Scenarios and Metrics of Interest

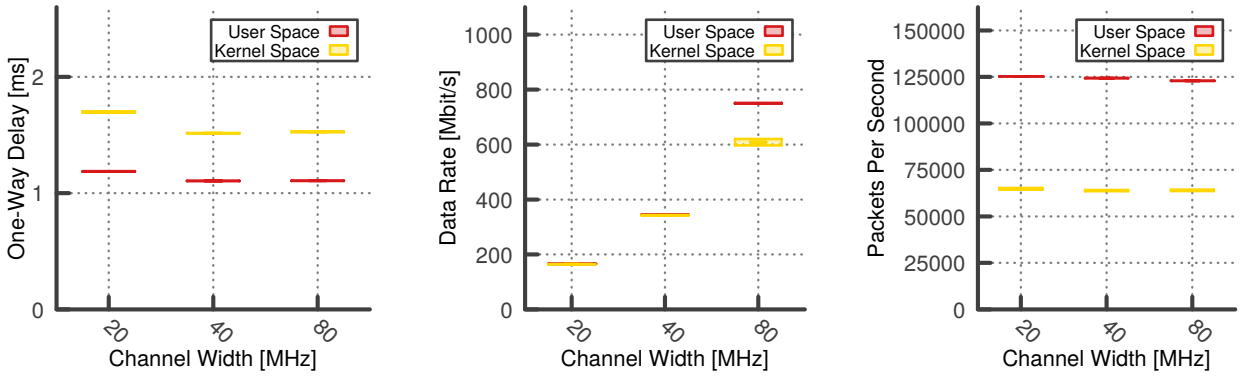
Any given scenario for each mode is evaluated using 32 repetitions for statistically significant results. The scenarios cover practical use cases, as well as edge cases to explore performance limitations of our setup. Over the course of our evaluation, we run every scenario listed below in both modes of operation (Sec. V-C) and compare different metrics. Those metrics are obtained either using our custom traffic generator or `iperf`. We explain for each scenario, which metric is of particular interest:

Scenario 1 — TCP Performance: This file-transfer-like scenario measures maximum TCP performance. Therefore, `iperf` is well-suited to generate TCP traffic in one direction to obtain the mean network throughput in Mbit/s.

Scenario 2 — Constant Bit Rate: One-way delay is of interest in this scenario. It resembles multimedia streaming. The transmission rate is set to 50,000 PPS – which is feasible for both modes of operation using packets of 60 Byte. Small packets also yield more individual delay measurements. For measurements, our custom tool for traffic generation will be used.

Scenario 3 — Maximum Data Rate: This scenario tries to explore the maximum data rate possible for both modes of operation. Therefore, the load generator is configured to send out large packets (1500 Byte) at 1 Gbit/s wire speed. In contrast to scenario 1, UDP is employed and therefore, no transport layer mechanisms are limiting the resulting data rate.

Scenario 4 — Maximum PPS: For that scenario, we focus on transmitted PPS. It evaluates the overhead imposed by processing a single packet for each mode of operation much better than the former scenario does, as small packets imply a larger possible quantity of transmitted packets and thus, more processing overhead. Strictly speaking, the scenario already resembles properties of a denial-of-service attack, as the wired data rate is too high to be forwarded over the wireless link and using small packets puts more stress on hard- and software than usual.



(a) Scenario 2 – Achieved delay.

(b) Scenario 3 – Achieved data rate (1500 Byte pkts.).

(c) Scenario 4 – Achieved PPS (60 B pkts.).

Fig. 4. Measurement results (99% confidence intervals).

E. Quantitative Results

First, we look at scenario 1 and the results concerning TCP performance in each mode. The general conjecture is that our approach is at least as fast as kernel mode. Table II summarizes the results of our measurements for scenario 1.

TABLE II
SCENARIO 1: TCP PERFORMANCE, INCL. CONF. INTERVAL (99%).

Channel Width	Kernel Mode [Mbit/s]	User Space [Mbit/s]
20 MHz	128.4 ± 1.1	135.3 ± 1.2
40 MHz	253.5 ± 2.0	268.5 ± 1.3
80 MHz	474.8 ± 6.2	524.6 ± 6.0

We can see that our approach outperforms kernel-mode operation for each examined channel width. Confidence intervals are tight and lead to statistically significant improvements of 5%, 6%, and 10% (for 20, 40, and 80 MHz channels respectively) when using our user-space implementation. Whilst performance equal or close to kernel-mode operation would be sufficient, TCP data rate improvements up to 10% are positive results.

Scenario 2 depicts a test setting for measurements of one-way delay. The conjecture is that our user-space approach yields smaller delays, as the polling-based architecture constantly checks for work to be done – hereby omitting kernel context switches in case packets need to be processed.

Fig. 4a illustrates results of our delay measurements. For each tested channel width, a decrease in one-way delay of at least 0.4 ms (27%) meets our expectations. It is noteworthy that these improvements arise from frame processing of two wireless boxes due to the experimental setup depicted in Fig. 3 (and the designated packet flow), as well as from higher performance on wireless and wired interfaces in user-space mode. However, additional benchmarks (not included in this evaluation) revealed that less than 60 μs of the difference in one-way delay originate from improvements of wired interface performance and are therefore relatively small.

Moving on to scenario 3, we are interested in the maximum data rate that can be forwarded for each mode of operation using large packets. Additionally, we compare performance with respect to transmitted PPS using small packets (60 Byte) for scenario 4. In the latter case, the overhead for single packet processing should dominate the resulting performance and that allows conclusions whether or not our approach to omit context switches in case of wireless frame processing increases performance.

Results for scenario 3 can be seen in Fig. 4b. Both approaches are of equal performance for 20 MHz and 40 MHz channels. As the per-packet processing overhead will not limit the performance (due to large packets), the channel width seems to confine the data rate equally for both user-space and kernel-space approach (as confidence intervals are overlapping). For 80 MHz however, this limitation seems to be overcome, as we can see that our user-space approach transmits significantly more packets (23%) than the kernel – already indicating that the per-packet processing overhead is smaller with our user-space approach.

When using small packets for scenario 4, the overhead for packet processing will be the only limiting factor. Fig. 4c reveals that our approach almost doubles the performance for any given channel width when comparing it to kernel mode operation – hereby exceeding our expectations.

F. Discussion of Performance Difference to Linux IEEE 802.11 Stack

To identify all components and functions in which the kernel spends the most time, the Linux perf subsystem in combination with flame graphs [18] was utilized. Flame graphs depict time-annotated system-wide call graphs. We used them to analyze the kernel's mac80211 tx path[1] on wireless boxes in the formerly introduced setup (Fig. 3). Measurements were taken for 20 seconds of unidirectional 100,000 PPS, 64 Byte in size (load on the wired interface), while handling roughly 50,000 PPS over

[1] *Flame Graph Image:* https://raw.githubusercontent.com/telematik-tu-ilmenau/DPDK-WiFi/master/data/kernel_80211_tx.svg

the wireless interface. A flame graph of the rx path was also created with the same method. We focus on the tx path in the following, as it is the more complex case due to Quality of Service (QoS) queues, IEEE 802.11 header creation, retransmit handling, and partial skb reallocation for IEEE 802.11 header insertion.

Main kernel components of relevance in this experiment are: ath10k wireless driver, igb wired driver, mac80211 subsystem, and bridge module. Therefore, performance differences possibly result from:

- P1 Interrupt handling vs. polling
- P2 Wired driver
- P3 Bridge implementation
- P4 Data structures of network stack and algorithms (e.g. skb, idr[2])
- P5 Mac80211 subsystem
- P6 Ath10k driver

In both rx and tx path, New API (NAPI) poll mode is used. Therefore, P1 should not cause any difference. P2 can also be ruled out, as the e1000 driver in DPDK and the igb driver in the Linux kernel share a large amount of code. We omitted a generic bridge implementation for our user-space benchmarks and move frames between the interfaces with a small layer of specialized code. Forwarding (P3) utilizes 9% of CPU time in our experiment. 62% of the CPU time is used for the wireless transmission, 34% in the mac80211 subsystem (P5), and the rest in the ath10k driver (P6).

The wireless transmission codepath consists of the following main steps:

1) Header creation (`ieee80211_build_hdr`)
   - Mesh path lookup
   - Fill in addresses after lookup
   - Set QoS header
2) Frame transmission (`ieee80211_xmit`)
   - Expanding or reallocation of frame header data (`pskb_expand_head`)
   - `ieee80211_tx`
     - Dequeue frame and handle Controlled Delay (CoDel) Active Queue Management (AQM)
     - Allocate transmission context id (`idr_alloc`, allows for frame deletion after finished (re-)transmission)
     - Send transmission context via DMA to hardware (`ath10k_pci_hif_tx_sg`)

The idr data structure, used for the allocation of transmission IDs, stores its data in a radix-tree-like fashion. Tree-based data structures provide a good compromise between size and performance, but need performance-degrading dynamic memory allocations. Our DPDK-based implementation uses two large arrays for ID allocation. The first array stores a free bitmask of the second one. This structure potentially uses more memory, but is more cache friendly and omits additional memory allocations. By using type propagation through C++ templates, our code uses less indirect pointer accesses. We preallocate large headroom in DPDK's mbufs in order to omit copy operations induced by the kernel's `pskb_expand_head`. The ath10k Linux kernel driver aggregates subframes before handing them to the mac80211 subsystem. In our code, subframes are enriched with statistics and passed around in an array of `rte_mbuf` pointers, without further aggregation. All in all, our approach outperforms the kernel thanks to a specialized implementation of P3, P4, P5, and P6.

[2] *idr*: Linux kernel data structure for integer id allocation and access to associated data, based on a radix-tree-like structure.

VI. Conclusion and Future Work

To cope with novel challenges, such as higher bandwidths and more complexity, software processing of 802.11 frames will strongly benefit from a new architecture. Besides the ability to deliver high performance and flexibility, this user-space approach allows fellow researchers to build more realistic wireless prototypes and experiments. This paper presented an architecture that incorporates wireless frame processing into a well-known framework for fast packet I/O (DPDK). This allows for easier and more rapid development (compared to kernel solutions), higher performance, and seamless integration of wireless interfaces into applications that already use user-space processing for wired interfaces. In particular, we incorporated a driver for ath10k family interfaces, for which we were able to show that it outperforms kernel-mode operation concerning delay and data rate. We also gave advice on porting (other) wireless drivers from kernel into user space. The code of our Wi-Fi enhanced DPDK version is available at Github[3]. The repository also contains an example application demonstrating the usage of the Wi-Fi layer. With the architectural blueprint in this paper, it is possible to build high-speed wireless APs or MPs simply by using small embedded CPUs, already using low energy. Further research is needed on the performance impact of additional power saving mechanisms, like clock scaling or interrupt handling in low utilization periods.

We work on upstreaming the architecture as part of DPDK and hope for participation of the research community to incorporate more wireless hardware and drivers. In future, we plan to investigate cross-layer optimization, robust routing, and secure end-to-end communication in WMNs using our architecture, as well as its integration in Network Function Virtualization (NFV) environments. Furthermore, we are working on a converged system that allows the incorporation of real-world software and simulations to rapidly evaluate network services.

[3] https://github.com/telematik-tu-ilmenau/DPDK-WiFi

REFERENCES

[1] D. Chen, J. Schuler, P. Wainio, and J. Salmelin, "5G Self-optimizing Wireless Mesh Backhaul," in *INFOCOM WKSHPS*, 2015.

[2] F. Samie, L. Bauer, and J. Henkel, "IoT Technologies for Embedded Computing: A Survey," in *CODES+ISSS*, 2016.

[3] L. Rizzo, "netmap: A Novel Framework for Fast Packet I/O," in *USENIX ATC*. Boston, MA: USENIX Assn., 2012.

[4] fd.io – Linux Foundation. Technology – The Fast Data Project. Accessed: 11-Jan-2018. [Online]. Available: fd.io/technology

[5] H. Wippel, "DPDK-based Implementation of Application-tailored Networks on End User Nodes," in *NOF*, 2014.

[6] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The Design and Implementation of Open vSwitch," in *Networked Systems Design and Implementation*, 2015.

[7] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the Art of Network Function Virtualization," in *Networked Systems Design and Implementation*. Seattle, WA, USA: USENIX Assn., 2014.

[8] G. Pongrácz, L. Molnár, and Z. L. Kis, "Removing Roadblocks from SDN: OpenFlow Software Switch Performance on Intel DPDK," in *Workshop on Software Defined Networks*, 2013.

[9] I. T. Haque and N. Abu-Ghazaleh, "Wireless Software Defined Networking: A Survey and Taxonomy," *IEEE Communications Surveys Tutorials*, vol. 18, no. 4, 2016.

[10] T. Lin, H. Bannazadeh, and A. Leon-Garcia, "Introducing Wireless Access Programmability using Software-Defined Infrastructure," in *IFIP/IEEE IM*, 2015.

[11] ntop di Deri Luca. PF_RING. Accessed: 09-Jan-2018. [Online]. Available: ntop.org/products/packet-capture/pf_ring

[12] Linux Foundation. Data Plane Development Kit. Accessed: 09-Jan-2018. [Online]. Available: dpdk.org

[13] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser, "Automatic Device Driver Synthesis with Termite," in *Symposium on Operating Systems Principles*. ACM, 2009.

[14] V. Chipounov and G. Candea, "Reverse Engineering of Binary Device Drivers with RevNIC," in *EuroSys*. ACM, 2010.

[15] S. Boyd-Wickizer and N. Zeldovich, "Tolerating Malicious Device Drivers in Linux," in *USENIX ATC*. Boston, 2010.

[16] P. Dely, A. Kassler, and N. Bayer, "OpenFlow for Wireless Mesh Networks," in *ICCCN*, July 2011.

[17] OpenWrt Project. Welcome to the OpenWrt Project. Accessed: 19-Feb-2018. [Online]. Available: openwrt.org

[18] B. Gregg, "The Flame Graph," *Commun. ACM*, vol. 59, no. 6, pp. 48–57, May 2016. [Online]. Available: http://doi.acm.org/10.1145/2909476