Understanding Linux Kernel-based Packet Switching on WiFi Access Points

Shiqi Zhang, Mridul Gupta, and Behnam Dezfouli
Internet of Things Research Lab, Department of Computer Science and Engineering, Santa Clara University, USA
{szhang9, magupta, bdezfouli}@scu.edu

***Abstract*—As the number of WiFi devices and their traffic demands continue to rise, the need for a scalable and high-performance wireless infrastructure becomes increasingly essential. Central to this infrastructure are WiFi Access Points (APs), which facilitate packet switching between Ethernet and WiFi interfaces. Despite APs' reliance on the Linux kernel's data plane for packet switching, the detailed operations and complexities of switching packets between Ethernet and WiFi interfaces have not been investigated in existing works. This paper makes the following contributions towards filling this research gap. Through macro and micro-analysis of empirical experiments, our study reveals insights in two distinct categories. Firstly, while the kernel's statistics offer valuable insights into system operations, we identify and discuss potential pitfalls that can severely affect system analysis. For instance, we reveal how packet switching rate and the implementation of drivers influence the meaning and accuracy of statistics related to packet-switching tasks and processor utilization. Secondly, we analyze the impact of the packet switching path and core configuration on performance and power consumption. Specifically, we identify the differences in Ethernet-to-WiFi and WiFi-to-Ethernet data paths regarding processing components, multi-core utilization, and energy efficiency.**

***Index Terms*—802.11, Linux, Monitoring, Measurement, Power Consumption, Processor Utilization, Function Tracing.**

I. Introduction

The advent of new applications and services, particularly those involving high-definition streaming, edge and cloud computing, and increasingly sophisticated Internet of Things (IoT) devices, necessitate the enhancement of WiFi technology in terms of rate, reliability, and scalability [1]. Parallel to this need for speed and efficiency, there is a significant increase in the number of WiFi Access Points (APs), which are mainly responsible for the implementation of the data plane for switching packets between their Ethernet and WiFi Network Interface Cards (NICs). Statistics show that the gigabit AP market alone is projected to increase at a compound annual growth rate of about 32.3% between 2024 and 2034 [2]. This upsurge is a response to the expanding reach of Internet connectivity and the need for high-speed and reliable coverage in various spaces.

Compared to enterprise and datacenter switches, which implement their data plane functions in hardware for high-speed processing, APs often rely on the Linux kernel for packet switching tasks [3]–[5]. This architectural difference is primarily due to the differing packet switching rates: modern enterprise switches handle immense data traffic, often exceeding several Tbps, while APs generally manage lower rates, typically in the range of Mbps or Gbps. This lower switching rate is due to their focus on providing wireless network access rather than core data routing. Consequently, using the Linux kernel for software-based packet switching in APs offers a cost-effective and flexible solution that meets their data throughput requirements.

Given the increasing complexity of operations and the growing number of APs deployed in diverse environments, understanding and improving packet switching performance on these devices has become increasingly necessary. This importance stems from several factors. First, as APs are being integrated into a broader range of application domains, such as edge computing, IoT, AR/VR, and real-time systems, the need to customize the data plane to support flow-level analysis and advanced traffic management techniques is becoming more important [6]–[9]. Although some APs incorporate hardware-accelerated data planes (e.g., [10]), software-based switching remains essential for enabling programmability and flexibility in packet processing. Software switching allows for the implementation of deep packet inspection, per-flow access control, anomaly detection, and forwarding traffic to containers running on AP [11]–[13]. These capabilities are critical as APs evolve into policy enforcement points in home, enterprise and IoT networks [6], [14]. Furthermore, with the increase in bandwidth and latency demands from associated stations, implementing traffic shaping and bandwidth slicing techniques [9], [15]–[19] becomes more important—tasks that are significantly easier to realize in software-based switches. Second, software-based packet switching is subject to performance variability due to processor scheduling, interrupt handling, and contention for shared resources [20]–[23]. Unlike hardware pipelines with deterministic latency, software switching operates within the general-purpose processing environment of the operating system. As a result, factors such as interrupt affinity, SoftIRQ scheduling, and queuing discipline (qdisc) behavior can directly impact processor utilization, forwarding jitter, and throughput. Third, collecting and interpreting statistics from the data plane is essential for identifying bottlenecks and improving overall system performance. Such statistics can also be leveraged by machine learning methods to dynamically tune system parameters [6], [24]–[27]. However, when the semantics and validity domains of these statistics are not well understood, their effectiveness and reliability as input features for learning algorithms may diminish.

Despite the growing importance of software-based switching challenges and performance analysis in APs, existing research has largely focused on high-performance environ-

ments, including resourceful servers [20], [22], [24], [28]. Furthermore, given the superior performance of kernel bypass methods like Data Plane Development Kit (DPDK), existing studies emphasize these technologies [19], [24], [29], [30] for packet switching between Ethernet NICs and user-space components such as Virtual Machines (VMs) and containers. In contrast, APs have distinct requirements, primarily performing packet switching between Ethernet and WiFi NICs.

In this paper, we study and analyze the operation and performance of two data paths, namely the **Ethernet to WiFi (E2W)** and **WiFi to Ethernet (W2E)** packet switching paths, on WiFi APs, focusing on two main questions: (i) *How and to what extent can the statistics provided by Linux and its user-space performance monitoring tools be used to understand packet switching operation and performance?* (ii) *What are the differences and their causes in processing resources and power consumption between E2W and W2E packet switching?* To address these questions, we first provide an overview of the primary operations and stages of packet switching in the latest Linux kernel. Next, given that many commercial APs rely on ARM processors for packet switching [31], we use two ARM-based platforms—with different types of Ethernet and WiFi interfaces—to empirically study and analyze packet switching operations. Integral to this empirical analysis, we perform both macro- and micro-analysis of packet switching operations to understand the performance characteristics and resource consumption patterns unique to WiFi APs. The scripts, measurement tools, and selected datasets used in this study are available at our GitHub repository [32].

Our main findings are as follows. (i) **Despite the rich set of statistics provided by Linux regarding the number of Interrupt Requests (IRQs) and SoftIRQs, interpreting these metrics requires an understanding of the Linux kernel's operation, system configuration, system load, and the specifics of both hardware and software implementations.** As a generic observation, we show that the number of RX SoftIRQs is influenced not only by the processing of incoming packets but also by the number of TX IRQs. As a device-specific observation, we show that although the Linux networking subsystem defines a protocol for transitioning between New API (NAPI) (polling) and IRQ-based operations, device drivers can override this protocol, thereby altering the semantics of the provided statistics. (ii) **The processor cycles consumed within the interrupt handling context are not accurately accounted for and reflected in processor utilization statistics.** For instance, the problem is observed for SoftIRQ instances handled immediately following Top Half (TH) interrupt processing and before the execution of the `ksoftirqd` thread. In this case, the execution of these SoftIRQ instances is incorrectly accounted for as part of `idle` processor utilization instead of SoftIRQ utilization. (iii) **Even when the two NICs are assigned to separate cores, packet switching in the E2W path does not utilize multi-core processing.** This limitation can result in a throughput drop if the core assigned to the Ethernet NIC cannot handle the maximum supported throughput of the two NICs. In contrast, packet switching in the W2E path utilizes two cores. (iv) **The processing load and power consumption of the E2W path are higher than those of the W2E path. However, as the throughput reaches the maximum supported levels, the differences in efficiency between the two paths reduce.**

To frame the context of our study, we outline two key considerations. First, although our study focuses on software-based packet switching, we recognize that some AP platforms may incorporate hardware-accelerated data planes. Second, given the widespread adoption of ARM-based processors and the SoftMAC architecture by APs (cf. Sections III, IV-C, and VI), our analysis is conducted using two Linux distributions (Ubuntu 6.7 and OpenWrt 6.6) on such platforms. While our objective is to uncover generalizable trends and overarching principles, it is important to note that updates to the Linux kernel or the use of FullMAC (instead of SoftMAC) WiFi devices may lead to different outcomes.

The rest of this paper is organized as follows. Section II provides an overview of packet processing stages in the Linux kernel. Testbed components and the collection of performance metrics are detailed in Section III. We present a macro-analysis of packet switching metrics in Section IV-A. In Section V, we provide an in-depth, micro-analysis of the packet switching operations. Section VI presents related work and future directions. We conclude the paper in Section VII.

II. An Overview of Linux Kernel Packet Switching

In this section, we provide an overview of the steps involved in the Linux kernel's data path for switching packets received on an ingress Network Interface Card (NIC) to an egress NIC, based on our analysis of the latest Linux kernel source code (version 6.7). Figure 1 illustrates the major steps in the process of packet switching from NIC 1 (ingress) to NIC 2 (egress).

A. Ingress Side Processing

NIC to RX buffer. When a network packet arrives at a NIC, it is first stored in the NIC's internal buffers. The NIC then uses Direct Memory Access (DMA) to transfer packets to the system's main memory, specifically into the buffers managed by the driver, updating the receive (RX) buffer's pointers in the process. The RX buffer is a ring queue, facilitating producer (NIC) and consumer (driver) operation. This action triggers an Interrupt Request (IRQ), invoking the corresponding interrupt handler, often referred to as the TH. The Top Half (TH) acknowledges the interrupt, performs preliminary handling, and then schedules further processing in a deferred manner, referred to as the Bottom Half (BH).

Generating an interrupt for each packet reception results in a high interrupt processing overhead. New API (NAPI) addresses this problem by utilizing a *polling* mode, where the driver periodically checks if new descriptors have been used for packet reception. Within this method, processing a TH involves calling the `napi_schedule` function of the kernel to start the NAPI for the RX buffer. To this end, when processing the TH for an IRQ, the `napi_schedule` function is called with a `napi_struct` parameter that includes a pointer to the poll function of the driver. Afterwards, the TH calls the `__raise_softirq_irqoff` function to raise a SoftIRQ, specifically an RX SoftIRQ designated for processing received

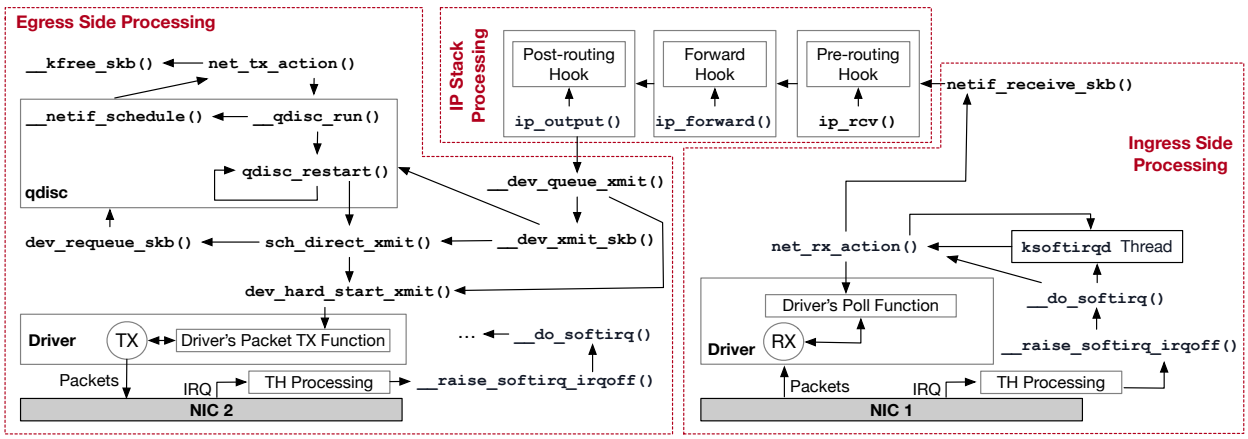

Fig. 1. The main steps of packet switching. Here, we assume the ingress interface is NIC 1 and the egress interface is NIC 2.

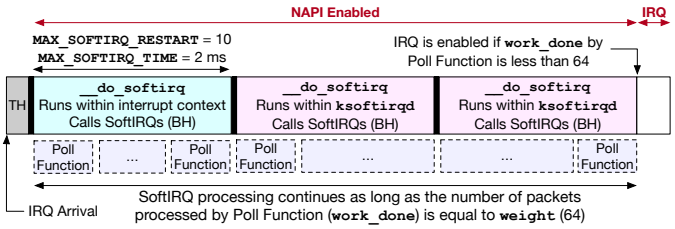

Fig. 2. Switching between NAPI and IRQ. NAPI continues as long as the poll function fully utilizes its packet processing budget.

packets. At this point, BH execution of ingress processing starts.

SoftIRQ execution. If there is any pending SoftIRQ, the kernel's `__do_softirq` function handles RX SoftIRQs by calling the `net_rx_action` function, which processes the network packets. The `net_rx_action` function utilizes the per-core `softnet_data` structure to access and iterate over the `napi_struct` list. Each entry of this list represents a network device with packets ready to be processed. The function dequeues each structure in turn and runs their driver's poll functions. Note that *the SoftIRQ is run on the same core that ran the TH, i.e., the core that received the IRQ*. After processing the TH, the `__do_softirq` function can immediately run within the IRQ context, subject to certain limits on the number and duration of the SoftIRQs called. Once the limits are exceeded, if more SoftIRQs must be run, the `__do_softirq` function is deferred to a kernel thread (`ksoftirqd`) or a tasklet.

To better understand the execution context of the `__do_softirq` function, we present Figure 2. When an IRQ's BH is triggered, the initial processing is handled by immediate RX SoftIRQ processing within the `__do_softirq` function. This immediate processing, *which runs within the interrupt context*, is essential for rapidly and efficiently handling incoming network traffic. The `__do_softirq` function operates under two main constraints. The first constraint, defined by the variable `MAX_SOFTIRQ_RESTART`, is the maximum number of consecutive times the SoftIRQ handling loop can execute the driver's poll function without yielding control back to the system. The second constraint, defined by the variable `MAX_SOFTIRQ_TIME`, specifies the maximum duration the kernel can spend processing SoftIRQs in a single invocation of `__do_softirq`. When the packet switching load is high, not all packets can be processed within the interrupt context while enforcing the above constraints. Works exceeding these immediate processing constraints are deferred to the kernel thread or a tasklet, depending on the device driver implementation. During handling such deferred works, as long as 64 packets are processed per poll function invocation, the NAPI poll is rescheduled, and the NIC IRQ remains disabled.

Switching between NAPI and IRQ. The contract between the NAPI subsystem and device drivers includes an important aspect related to the deactivation of NAPI. Every time a driver's poll function is called, the number of processed packets is returned in the `work_done` variable. If a driver's poll function consumes its full `weight` allotment (set as 64 by default), the NAPI state stays unchanged, and control is returned to the `net_rx_action` loop, which may call the poll function again if the processing time limit allows. Conversely, if the poll function does not use its entire weight, it must disable NAPI. Subsequently, NAPI will be reactivated upon receipt of the next IRQ; at this point, the driver's IRQ handler is expected to invoke the `napi_schedule` function.

Delivery to IP stack. The `netif_receive_skb` function is responsible for further packet processing and delivery to the IP stack.

B. IP Stack Processing

For IP packets, the function `ip_rcv` is called. This function calls the *pre-routing* hook of the Netfilter subsystem. If this hook does not drop the packet, the routing subsystem is consulted to determine its destination. If the packet is meant for another system, the `ip_forward` function is called to determine the packet's next hop by consulting the routing table. Subsequently, the packet may be modified and evaluated against firewall rules within the *forward* Netfilter hook. The kernel performs further routing decisions after passing the forward hook. This involves consulting the routing table to determine the next-hop address and the appropriate outgoing

network interface for the packet. Afterward, various IP-level checks and updates occur. If the packet is modified (e.g., TTL is decremented), the IP header checksum is recalculated. As the packet is almost ready to be sent out, it approaches the *post-routing* hook. This hook is the last chance to inspect and modify the packet before it leaves the system.

C. Egress Side Processing

One of the main components of egress side processing is qdisc. In Linux, qdisc (short for queuing discipline) is a mechanism used to control how packets are queued and transmitted. The default qdisc in most Linux distributions is pfifo_fast, which implements a simple First-In First-Out (FIFO) queue with three bands of traffic priorities. Another algorithm is FQ-CoDel [33], which is a modern qdisc designed to combat buffer-bloat by managing queue lengths and ensuring fair bandwidth distribution. Although Ethernet NICs utilize a qdisc on their egress direction, in Sections IV and V we will discuss the lack of qdisc on the egress direction of WiFi NIC and its implications on performance.

IP stack to driver. After IP stack processing, the packet is passed to the netdev (short for network device) subsystem using the `__dev_queue_xmit` function. If there is a qdisc associated with the interface, this function passes the packet to the `__dev_xmit_skb`. Otherwise, the packet is directly added to the transmit (TX) buffer (a ring queue) of the driver by using the `dev_hard_start_xmit` function.

If there is a qdisc associated with the interface, the function `__dev_xmit_skb` first acquires a lock on the qdisc, and then checks if the packet can bypass the qdisc under certain conditions, such as when the qdisc is empty. If these conditions are met, the function attempts to transmit the packet directly, via the `sch_direct_xmit`, *bypassing the usual queuing mechanisms for efficiency.* If `__dev_xmit_skb` cannot bypass the qdisc, the packet is enqueued in the qdisc. Then, if the qdisc is not running, the function `__qdisc_run` is called to run it. As long as there are packets in the qdisc and the quota of running qdisc has not been fully used, `__qdisc_run` calls `qdisc_restart` in a loop to dequeue packets from the qdisc and send them to the TX buffer. When the quota is exhausted, the `__netif_schedule` function is called to schedule a TX SoftIRQ, which is used for processing outgoing network packets by running the `net_tx_action` function. This function accesses the `softnet_data` structure of *the core to which the IRQ of the egress NIC is assigned affinity*, and checks if the TX buffer has any SKBs that must be freed (using the `__kfree_skb` function). Then, if there are more packets in the qdisc, the function `__netif_schedule` runs the qdisc again.

To pass a packet to the driver, a lock is acquired by the `dev_hard_start_xmit` function on the driver's TX buffer to prevent concurrent access to the transmit queue by other cores. After attempting to transmit the packet, the transmission lock on the TX buffer is released. The function then checks if the transmission was complete. If the transmission was not successfully completed (e.g., if the network driver did not successfully validate SKB), the packet is requeued using `dev_requeue_skb` to attempt transmission again later.

Driver to NIC. The driver allocates buffers in the system's RAM, from which the DMA engine can read the data and transfer to the NIC. Once the DMA is set up, the driver triggers the network device to initiate the transmission. This usually involves writing to specific device registers, indicating the readiness of a packet for transmission and providing the DMA-prepared memory buffers' locations. Once the NIC completes the packet transmission, it signals this completion by generating a TX IRQ to inform the driver that the transmission has finished. The IRQ results in calling TH and then BH to perform several critical post-transmission tasks. It starts by unmapping any DMA mappings that were previously established, ensuring proper memory management and preventing resource leaks. Following this, the driver frees the memory buffers allocated for the packet, typically involving the deallocation of the associated SKBs.

III. Testbed Components and Collection of Performance Metrics

In this section, we elaborate on the AP platforms used in our studies, the testbed configuration, and data collection tools.

A. AP Platforms and the Testbed

For the empirical evaluations in this paper, we used two different Access Point (AP) platforms. The first platform is based on the Raspberry Pi (RPi) Compute Module 4 (CM4) SoC, which features a quad-core ARM Cortex-A72 processor running at 1.5 GHz. It employs the BCM54210PE [34] Ethernet controller, which supports 1 Gbps and is driven by the bcmgenet driver. This platform also includes a PCIe 2.0 x1 host controller, which we used to connect an Intel AX210 WiFi 6E (802.11ax) NIC. The theoretical maximum throughput of the WiFi NIC is 2.4 Gbps. Since this platform supports WiFi 6, we refer to it as the **APW6** platform in this paper. The APW6 platform runs Ubuntu 6.7. The second AP platform is based on MediaTek's Filogic 880 SoC, which features a quad-core ARM Cortex-A73 processor running at 1.8 GHz. This platform uses the MediaTek MT7986A Ethernet controller, managed by the mtk_eth_soc driver, and supports 1 Gbps and 10 Gbps connectivity. In this paper, we use the 1 Gbps connectivity. The SoC incorporates a PCIe 3.0 x1 host controller to interface with a MediaTek BE14 WiFi NIC, which supports WiFi 7 and is driven by the mt7996 driver. Given its support for WiFi 7, we refer to this platform as the **APW7** platform in this paper. The APW7 platform runs OpenWrt 6.6. We selected these two platforms because many Commercial Off-The-Shelf (COTS) APs are based on ARM Cortex-A processors [31]. All experiments were conducted using the 6 GHz WiFi band in an interference-free environment.

Figure 3 shows the testbed's components and their connectivity.

The AP (APW6 or APW7) is the device under test, based on the platforms mentioned above, running Linux kernel version 6.7. The Ethernet NIC of the AP is connected to Machine 1, while the WiFi NIC of the AP is connected to Machine 2.

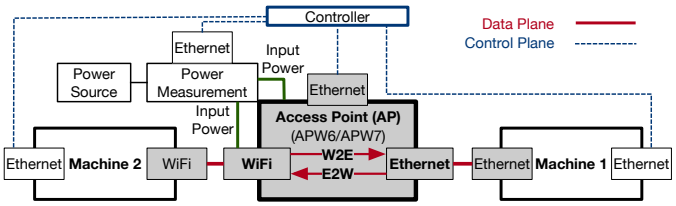

Fig. 3. The components, data plane, and control plane of the testbed. Data plane is used for data flow transmissions, and control plane is used for automation of experiments and collecting performance monitoring data.

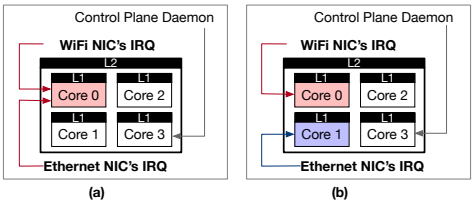

Fig. 4. (a) Single-core configuration: The IRQs of both NICs are assigned to Core 0. (b) Dual-core configuration: The IRQ of the WiFi NIC is assigned to Core 0 and the IRQ of Ethernet NIC is assigned to Core 1.

The qdisc used with the Ethernet interface is FQ-CoDel [33]. These connections constitute the data plane of the testbed. MTU-sized data flows are generated by either Machine 1 or Machine 2, depending on the packet switching path.

Machine 1 and Machine 2 belong to different subnets; therefore, the AP performs layer-3 switching. We disabled Generic Receive Offload (GRO) because its efficiency highly depends on the number of flows and their traffic patterns [20]. To build a control plane for automating the experiments and performing data collection from various components of the testbed, a Controller is connected to the AP, Machine 1, Machine 2, and the power measurement tool. We added a USB Ethernet dongle to the AP to provide control plane connectivity. Using this dongle introduces a negligible (less than 7%) processing overhead on one core. We developed a control plane daemon to run on the AP and collect various types of performance data. The collected data are sent to the Controller every second. This daemon is statically assigned to Core 3 of the AP to avoid interfering with packet switching tasks, which are assigned to other cores.

B. Core Configurations

Since the processors of both APW6 and APW7 consist of four cores, we investigate the impact of different core assignment configurations on packet switching performance. To achieve this, we adjusted the IRQ affinity for the Ethernet and WiFi NICs. These configurations, referred to as *single-core* and *dual-core*, are illustrated in Figure 4. It is worth noting that on both AP platforms, the system enforces that the WiFi NIC's IRQ must be assigned exclusively to Core 0, while the Ethernet NIC's IRQ can be assigned to any core. Therefore, for the single-core configuration, Core 0 is the only feasible option for these platforms.

C. Monitoring and Data Collection

We collect various types of performance and operational data from the AP platforms. Linux provides a suite of performance evaluation statistics accessible through the `proc` file system. The `/proc/interrupts` file offers comprehensive details about each IRQ, including the frequency of IRQ arrivals. The `/proc/softirqs` file represents the number of times RX SoftIRQs and TX SoftIRQs have been invoked. We utilize ethtool to collect statistics such as the number of frames received and sent. We measure processor utilization using mpstat. We collect the number of cycles consumed per core using the perf utility. The power consumption of AP is measured using a programmable power monitoring tool capable of sampling voltage and current at 1000 samples per second [35]. To specifically analyze the power consumption of the AP's processor, we subtract the power consumption of the WiFi NIC from the power measurement results.

IV. Performance Analysis and Demystifying Statistics: A Macroalaysis Approach

In this section, we first examine the operation and performance of packet switching in the E2W path, followed by the W2E path. Subsequently, we compare the differences between the two paths. Specifically, we focus on statistics collected from the `proc` file system, as well as processor utilization, processor cycles, and power consumption.

A. Ethernet-to-WiFi (E2W) Packet Switching

In this section, we present and discuss the results of Ethernet to WiFi (E2W) packet switching. Unless mentioned otherwise, a UDP flow's packets are received via the Ethernet NIC and transmitted by the WiFi NIC. For APW6, the results for single-core and dual-core configurations are presented in Figures 5 and 6, respectively. Sub-figures (a) in Figures 5 and 6 present packet switching statistics. Sub-figures (b), (c), and (d) demonstrate the utilization of the cores, the number of cycles per core per second, and power consumption, respectively. In sub-figures (b), 'Processor' refers to the average processor utilization across all the cores. In all these sub-figures, the results are presented for three distinct throughput levels. We selected two preset throughput levels, specifically 100 and 500 Mbps, alongside the maximum throughput achievable by each configuration. Each maximum throughput level represents the threshold beyond which the rate of ingress packets to the platform exceeds the rate of egress packets from the platform.

Observation 1: *The number of RX IRQs and RX SoftIRQs is not directly correlated with the rate of incoming packet processing. While the trends in these statistics may be influenced by the type and configuration of the NIC and its driver, this is generally a platform-independent observation, rooted in the behavior of the Linux kernel's networking subsystem.*

In Figures 5(a) and 6(a), as the throughput increases from 100 to 500 Mbps, and further to the peak throughput, we observe that the number of Ethernet RX IRQs (denoted as

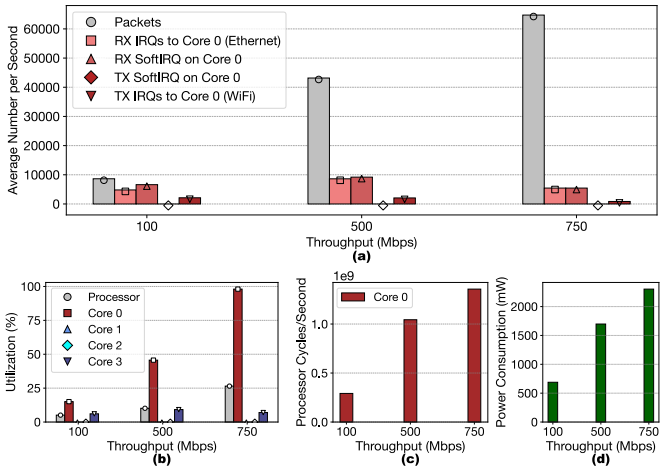

Fig. 5. Ethernet-to-WiFi (E2W) packet switching on the APW6 platform using the single-core configuration. The maximum achieved throughput of this configuration is 750 Mbps.

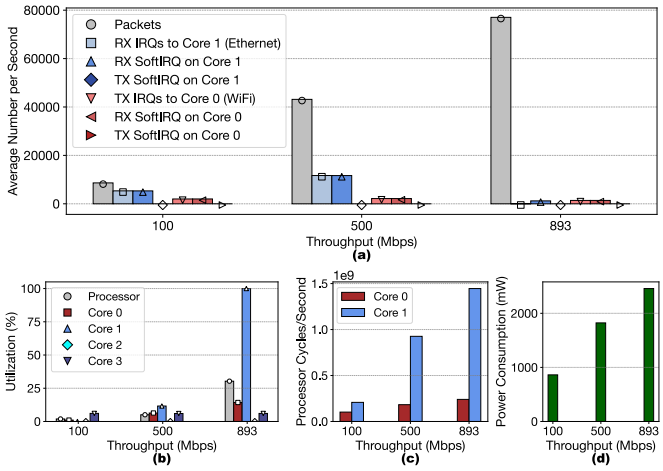

Fig. 6. Ethernet-to-WiFi (E2W) packet switching on the APW6 platform using the dual-core configuration. The maximum achieved throughput of this configuration is 893 Mbps.

square) rises and then begins to decline. Remarkably, this count drops to a near-zero value only for the dual-core configuration, as demonstrated in Figure 6(a).

With the dual-core configuration, a maximum throughput of around 893 Mbps is achieved, as shown in Figure 6(a). In this case, the number of Ethernet RX IRQs is approximately two per second, and the number of RX SoftIRQs on Core 1 is about 1140. Note that the count of RX SoftIRQs (in the `proc` file system) does not equate to the total number of packets processed. This is because each invocation of this SoftIRQ can process multiple packets as long as packets are present in the driver's RX buffer, until the processing quota is exhausted. To determine the number of packets processed per RX SoftIRQ invocation, we divide the number of packets processed per second by the number of RX SoftIRQs per second, resulting in $72700/1140 \approx 63.8$. This value is very close to the default NAPI `weight` of 64 packets (cf. Section II-A), leading us to conclude that each iteration of the NAPI poll processes the maximum number of packets and reschedules repolling of the RX buffer without re-enabling the Ethernet IRQ. Occasionally, when the number of packets processed falls below 64, the NAPI poll concludes and the Ethernet's IRQ is re-enabled, resulting in approximately two IRQs per second. The near 100% utilization of Core 1, as presented in Figure 6(b), confirms that this core is almost fully utilized by the NAPI polling mechanism. In particular, the high utilization of this core is caused by continuously renewing the NAPI and running the poll function within the `ksoftirqd` context, as discussed in Section II-A.

Observation 2: *The number of RX SoftIRQs is influenced not only by the processing of incoming packets but also by the number of TX IRQs. This observation is rooted in the behavior of the Linux kernel's networking subsystem.*

For the dual-core configuration of APW6, we observe in Figure 6(a) for 100 and 500 Mbps throughput levels that the number of RX SoftIRQs is the same as the number of RX IRQs on Core 1, confirming the activation of an RX SoftIRQ only when an Ethernet RX IRQ is triggered. However, for the single-core configuration, an interesting observation in Figure 5(a) is that the number of RX SoftIRQs is higher than the number of Ethernet RX IRQs, which is particularly evident for the 100 Mbps rate. We observed a similar behavior when using the APW7 platform. This behavior is surprising because when the throughput is low, the arrival of each Ethernet IRQ invokes an RX SoftIRQ, which can completely process the packets in the driver's RX buffer and exit without needing to renew the NAPI poll. Therefore, we expect the number of RX SoftIRQs to be the same as that of Ethernet RX IRQs.

We rationalize this observation as follows. First, note that there is one `softnet_data` structure per core (cf. Section II-A); therefore, for the single-core configuration, the IRQs received from both NICs contribute to the increase in the number of SoftIRQs on that core. Additionally, by reviewing the `netdev` subsystem of Linux [36] and using the `ftrace` utility [37], *we validated that each TX IRQ generated by a NIC (following the transmission of one or more packets) also triggers the activation of a RX SoftIRQ on the same core.* In Figure 5(a), when a TX IRQ is generated by the WiFi NIC, the IRQ handler performs two operations: it adds the driver's poll function to the list of NAPI polls for that core (Core 0), and then raises a RX SoftIRQ to call the driver's poll function. The driver's poll function is then responsible for performing transmission-completion tasks such as TX buffer reclaiming, which involves freeing descriptor entries and memory associated with packets that have been successfully transmitted.

Observation 3: *The Linux kernel may not correctly account for the processor cycles consumed within an IRQ context, depending on the implementation methodology of the driver. The severity of the misaccounting problem depends on the amount of work performed within the IRQ context rather than being deferred to a kernel thread or a tasklet.*

The results presented in sub-figures (b), (c), and (d) of Figures 5 and 6 for the APW6 platform indicate that the number of core cycles and power consumption exhibit similar

increasing trends; however, processor utilization does not follow the same pattern. For the single-core configuration, increasing throughput from 100 to 500 Mbps results in a 71% jump in utilization of Core 0, and for the dual-core configuration, increasing the throughput from 500 Mbps to the maximum value results in an 8% increase for Core 0 and 88% for Core 1. Another related observation can be made by comparing the actual values of processor utilization and the number of cycles between the single-core and dual-core configurations. Comparing Figures 5(b) and (c) with Figures 6(b) and (c) for the 500 Mbps throughput, we observe that while the total number of cycles (consumed by Core 0 and Core 1) for the dual-core configuration is higher than that of the single-core configuration (Core 0), the processor utilization of the single-core configuration is higher.

We present the quantitative results of this discrepancy in Table I for both APW6 and APW7. This table reports the *normalized core utilization* and *normalized processor cycles* consumed by the cores handling packet switching. Normalized core utilization is calculated by dividing the sum of the utilization of the cores involved in packet switching by the throughput level. For instance, on APW6, for the dual-core configuration, since the utilization values of Core 0 and Core 1 for the 500 Mbps throughput are 6.38% and 11.63%, respectively, the table reports $(6.38+11.63)/500 = 0.0360$ for this core's normalized utilization per 1 Mbps packet switching rate for the 500 Mbps throughput level. Normalized cycles consumed is calculated by dividing the sum of the percentage of the cycles consumed by the cores involved in packet switching by throughput level. Here, the percentage of cycles per core is calculated by dividing the number of consumed cycles by the core's frequency, which is 1.5 GHz per core for APW6 and 1.8 GHz per core for APW7. For instance, on APW6, for the dual-core configuration, since the cycles consumed by Core 0 and Core 1 for the 500 Mbps throughput are 182747814 and 926181648, respectively, the table reports $((182747814+926181648)/(1.5\times10^9))\times100/500 = 0.1479$ for this core's normalized cycles.

The highlighted cells in Table I indicate the normalized utilization values that are considerably lower than the corresponding normalized processor cycles. These values indicate that *the system does not report accurate processor utilization at these throughput levels*. To investigate the underlying cause of this inaccurate reporting, we *found that BH processing triggered by an incoming IRQ may not be correctly accounted for in the processor utilization metric*. We also observe that the degree of disparity between the reported processor utilization and the actual cycles consumed differs between APW6 and APW7. We elaborate on the causes and severity of this disparity in the following discussion.

In Section II-A, we explained that after the execution of a TH, the kernel schedules an RX SoftIRQ to process pending NAPI poll functions. This SoftIRQ is typically executed inline within the IRQ context as part of the SoftIRQ handling phase. However, when the restart limit or runtime limit of `__do_softirq` is reached, the remaining work is offloaded to the per-CPU `ksoftirqd` kernel thread, which completes the processing in a preemptible process context. For

TABLE I
NORMALIZED CORE UTILIZATION AND NORMALIZED CYCLES OF PACKET SWITCHING CORES USING APW6 AND APW7 PLATFORMS

E2W on APW6 using the the Single-Core Configuration	100 Mbps	500 Mbps	750 Mbps
Core 0 Utilization (%)	0.0127	0.1488	0.1307
Core 0 Cycles	0.1945	0.1392	0.1206
E2W on APW6 using the the Dual-Core Configuration	**100 Mbps**	**500 Mbps**	**893 Mbps**
Core 0 and 1 Utilization (%)	0.0105	0.0360	0.1280
Core 0 and 1 Cycles	0.2067	0.1479	0.1259
E2W on APW7 using the the Single-Core Configuration	**100 Mbps**	**500 Mbps**	**925 Mbps**
Core 0 Utilization (%)	0.1546	0.1161	0.1081
Core 0 Cycles	0.2466	0.1408	0.1080
E2W on APW7 using the the Dual-Core Configuration	**100 Mbps**	**500 Mbps**	**950 Mbps**
Core 0 and 1 Utilization (%)	0.1232	0.1286	0.1356
Core 0 and 1 Cycles	0.2435	0.1551	0.1351

the Ethernet interface of APW6, however, the RX SoftIRQ instances executed immediately after the TH and before the `ksoftirqd` thread is invoked are incorrectly accounted for as part of the *idle* processor utilization, rather than being attributed to SoftIRQ processing. We utilized two approaches to verify this *misaccounting* of SoftIRQ processor cycles on the APW6 platform. Firstly, using the `perf` and `ftrace` tools, we noticed that the functions of such RX SoftIRQs were recorded under the `idle` or `swapper` threads, and these cycles are added to the 'idle' category in `/proc/stat`. Therefore, tools such as `mpstat` misrepresent the actual processor utilization consumed by packet processing. Secondly, we modified the Linux kernel and changed the `MAX_SOFTIRQ_RESTART` value in the `__do_softirq` function from 10 (default) to 1. When running the modified kernel on APW6, this change decreased the number of times this function can rerun poll functions without invoking a `ksoftirqd` thread. Without this change, the utilization of Core 1 for the dual-core configuration was around 11% at 500 Mbps throughput, as Figure 6(b) shows. After applying the change to the kernel, the reported utilization on the same core increased to 67% (results are not shown in this paper). We now explain why the misaccounting problem is less severe on the APW7 platform. The Ethernet interface of APW7 uses a different mechanism for handling deferred work. In particular, in contrast to APW6, which relies on `ksoftirqd` to process deferred SoftIRQs, APW7's Ethernet interface delegates the work to a *tasklet*. With this approach, no packet processing is performed directly in the IRQ context; only tasklet scheduling occurs within that context. More specifically, while on the APW6 platform the misaccounted operations include both packet processing and the SoftIRQ-to-`ksoftirqd` transition handler, which wakes up the per-processor `ksoftirqd` thread, on APW7, the misaccounted component is limited to tasklet scheduling. It is also worth noting that we observed the misaccounting issue on the WiFi interface of the APW7 platform as well, as it similarly utilizes the tasklet mechanism for handling SoftIRQs. However, the

misaccounting problem was negligible on the WiFi interface of APW6, which uses threaded IRQ handling before delegating tasks to the `ksoftirqd` thread. This use of threaded IRQs results in more accurate accounting of work performed in the IRQ context. Therefore, the severity of the misaccounting problem is influenced by the amount of work performed directly in the IRQ handling context, as opposed to work deferred to a kernel thread or tasklet. In essence, the misaccounting behavior depends on the methodology used for handling IRQs and deferred work.

By revealing the main cause of the misaccounting problem, we provide additional details that justify the trends observed in Table I. Considering the APW6 platform, as throughput increases, the percentage of instances where Ethernet RX SoftIRQs run within the `ksoftirqd` context also increases, leading to more accurate accounting of processor cycles. Consequently, the misaccounting problem diminishes as throughput reaches its maximum value. For the dual-core configuration of APW6, as shown in Table I, the misaccounting problem persists at 100 and 500 Mbps throughput levels. This occurs because Core 1 processes an RX SoftIRQ per RX IRQ, as depicted in Figure 6(a). Therefore, RX SoftIRQs primarily run within the context of the IRQ handler, namely, the `idle` or `swapper` contexts, rather than `ksoftirqd`. In comparison, the single-core configuration exhibits the misaccounting problem only at the 100 Mbps throughput level because the percentage of RX SoftIRQs that need to run in the `ksoftirqd` context grows more rapidly than in the dual-core configuration. Additionally, we identify a secondary, more nuanced reason for this behavior. In the single-core configuration, each core has its own `softnet_data` structure (cf. Section II-A), which both NICs utilize. While the WiFi driver is reclaiming its TX buffer entries, an Ethernet RX IRQ may be generated, resulting in the addition of a NAPI instance for the Ethernet interface to the poll list of the same core's `softnet_data` structure. Within the `__do_softirq` function, after processing the WiFi driver's function, if the poll list of Core 0 is not empty, the core proceeds with running the poll function of the Ethernet driver within the context of a thread that has been created by the driver of the WiFi interface. Therefore, when the BH of the WiFi driver is running within a threaded interrupt context, it can execute Ethernet RX SoftIRQs. In this condition, the processor cycles consumed by the Ethernet RX SoftIRQs are properly accounted for under the SoftIRQ processing category. Considering the APW7 platform, increasing the throughput results in greater packet aggregation and a lower number of IRQs per second. As a result, the number of misaccounted tasklet scheduling instances decreases.

The discussions presented for this observation reveal that Linux's high-level processor utilization monitoring tools (e.g., `mpstat`) cannot be relied upon for accurate reporting. Instead, as demonstrated by the methodology presented in this section, the output of such processor monitoring tools, as well as more advanced, low-level tools that directly monitor processor cycles (e.g., `perf`), must be correlated and analyzed to identify the range of throughput values over which processor utilization is inaccurately reported.

B. WiFi-to-Ethernet (W2E) Packet Switching

In this section, we present and discuss the results of WiFi to Ethernet (W2E) packet switching path, where a UDP flow's packets are received via the WiFi NIC and transmitted by the Ethernet NIC. The results for APW6 are demonstrated in Figures 7 and 8. The methodologies for collecting and presenting results are the same as those described in Section IV-A.

Observation 4: *While Linux specifies a protocol for switching between NAPI and IRQ modes, drivers for certain NICs may override this protocol. Therefore, one cannot rely solely on NAPI-related statistics for a network interface without understanding the behavior and implementation of the underlying driver.*

In Section II-A, we explained the protocol for the transition between NAPI and IRQ. Building on this, we present a platform-specific case to demonstrate how this protocol can be overridden by device driver implementations. In the discussions of Observation 2 pertaining to the E2W path, we demonstrated that packet transmissions by an egress NIC trigger the execution of RX SoftIRQs, which perform tasks such as TX buffer reclaiming. For instance, for W2E packet switching on the APW6 platform, when a TX IRQ is generated, an RX SoftIRQ is scheduled to call the `bcmgenet_tx_poll` function of the `bcmgenet` driver to reclaim the space used by the transmitted packets. However, in Figure 7(a), the number of RX SoftIRQs exceeds the sum of Ethernet TX IRQs and WiFi RX IRQs, which is an unexpected observation for low throughput levels such as 100 Mbps. Additionally, in Figure 8(a), the number of RX SoftIRQs on Core 1 exceeds the number of TX IRQs on this core, which is unexpected for low throughput levels such as 100 Mbps. For example, for dual-core configuration at 100 Mbps, the average number of packets processed per RX SoftIRQ on the Ethernet (egress) side is $7651/2609 = 2.93$, which is much lower than the default `weight` value of 64 required to reschedule the RX SoftIRQ using the NAPI functionality. We analyze the operations of the `bcmgenet` driver's `bcmgenet_tx_poll` function to investigate this behavior. Typically, a poll function is expected to return the actual number of successfully sent (processed) packets as the `work_done` value.[1] However, the `bcmgenet` driver's `bcmgenet_tx_poll` function returns the kernel's default `weight` value of 64 as the `work_done`, as long as the actual `work_done` value is greater than 0. Thus, if at least one packet buffer is reclaimed, the NAPI is renewed to utilize an RX SoftIRQ to execute the `bcmgenet_tx_poll` function again. We speculate that this NAPI renewal method is employed to perform immediate post-transmission actions in anticipation of imminent upcoming transmissions. Therefore, by returning a value that does not reflect the actual number of processed packets, the driver overrides the NAPI agreement and alters the interpretation of statistics reported in the `/proc/stat` file system.

[1]For incoming packet processing, as explained in Section II-A and in Observation 1, the poll function returns the number of packets processed from the driver's RX buffer.

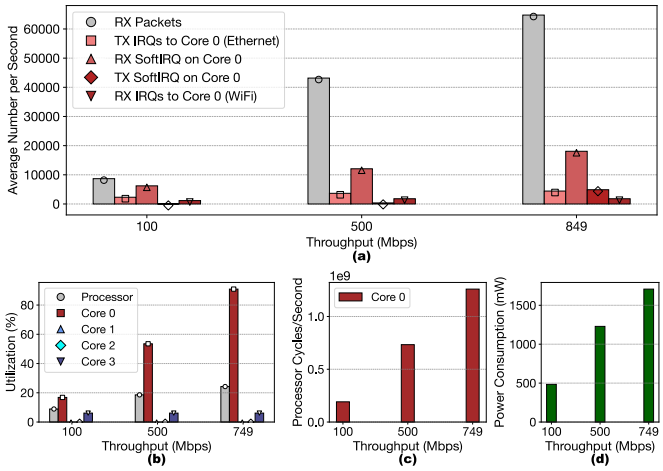

Fig. 7. WiFi-to-Ethernet (W2E) packet switching on APW6 using the single-core configuration. The maximum achieved throughput of this configuration is 849 Mbps.

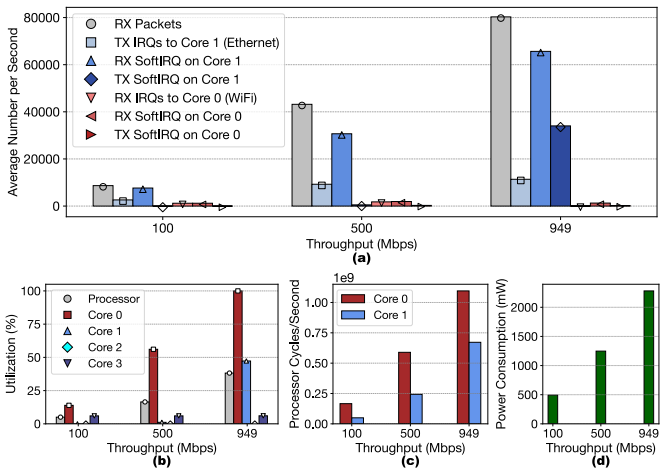

Fig. 8. WiFi-to-Ethernet (W2E) packet switching on APW6 using the dual-core configuration. The maximum achieved throughput of this configuration is 949 Mbps.

Observation 5: *If a network interface exhibits the processor cycles misaccounting problem when serving as the ingress interface, it also exhibits the same issue when serving as the egress interface.*

In the discussion of observation 3, we explained the processor cycle misaccounting problem when the Ethernet interface of APW6 or APW7 is performing ingress packet processing. The results presented in Figures 7 and 8 for APW6 reveal that the problem persists for the W2E path as well. For instance, comparing Figures 8(b) and (c), when the throughput is increased from 500 Mbps to the maximum value, the utilization of Core 1 increases by 4047%, whereas the increase in the number of cycles consumed by this core is 176%. Similarly, we observed the same behavior for W2E packet switching on the APW7 platform (results not shown in this paper). The reason for this behavior on APW6 is that when a TX IRQ is generated by Ethernet NIC for Core 1, the TX SoftIRQ instances executed on Core 1—immediately after TH processing and before the `ksoftirq` thread runs—are accounted as *idle* processor utilization. On APW7, the reason is that when a TX IRQ is generated by the Ethernet NIC for Core 1, the processing associated with tasklet scheduling is not accounted for in the processor utilization.

C. Comparison of E2W and W2E

Comparing the results for W2E and E2W paths, we observe significant differences in the performance and operation of packet switching for these two paths. We identify and justify these observations as follows.

Observation 6: *For W2E packet switching, using the dual-core configuration enables packet processing—from reception in the WiFi RX queue to placement into the Ethernet TX queue—to leverage both cores, thanks to the use of* qdisc *for the Ethernet interface. However, the extent to which the secondary core is utilized depends on the system load and the configuration of the egress NIC's TX queue. In contrast, for E2W packet switching, if the WiFi interface is based on the SoftMAC architecture, packet switching—from reception in the Ethernet RX queue to placement into the WiFi TX queue—is handled entirely by the core assigned to the IRQ of the Ethernet interface.*

By default, each network interface uses a qdisc as its entry point into the netdev subsystem, as discussed in Section II-C. The qdisc is essential for scheduling, prioritizing, and managing traffic flows to ensure efficient bandwidth utilization and reduce latency. However, to implement airtime fairness for downlink packet delivery, WiFi interfaces based on the SoftMAC architecture[2] employ the mac80211 kernel module, which integrates FQ-CoDel queue management with airtime fairness methods [3], [33], [38]. Consequently, using such devices, the qdisc module is completely bypassed on the WiFi interfaces. Note that the WiFi interfaces of both APW6 and APW7 are based on the SoftMAC architecture.

Referring to Figure 1, we explain the operation of E2W packet switching. When a packet received at the Ethernet interface reaches the `__dev_queue_xmit` function, it calls `dev_hard_start_xmit`, which forwards the packet to the mac80211 module for queuing and further processing before it is sent to the driver's TX buffer. After ingress Ethernet packet processing, the mac80211 packet processing continues within the interrupt or `ksoftirqd` context and runs on the core that is handling IRQs for the Ethernet interface; thereby, *packet processing from the ingress NIC to the egress NIC is handled by a single core, which is Core 1 in our testbed.* This also explains why the number of TX SoftIRQs for the WiFi interface is zero in the results presented in Figure 6(a).

[2]SoftMAC and FullMAC refer to two different WiFi architecture designs based on where the MAC layer functionality is implemented [38]. In a SoftMAC design, the MAC layer is handled by the host system, typically within the Linux kernel using the `mac80211` framework. This allows for greater flexibility, customization, and visibility into WiFi operations. In contrast, a FullMAC design offloads all MAC layer processing to the WiFi NIC's firmware, with the host system communicating through a simpler driver interface. This approach reduces processor load on the host and simplifies driver development, but is often less flexible and relies heavily on proprietary firmware.

Conversely, in the W2E path, a certain percentage of packets are added to the Ethernet interface's qdisc by the core processing incoming packets from the WiFi interface. Then the core handling the Ethernet IRQs performs qdisc processing and adds the packets to the TX buffer. More specifically, when the IRQs of the two NICs are assigned to different cores, whenever the TX buffer of the Ethernet interface's driver lacks space, Core 0 (responsible for handling ingress packets on the WiFi interface) adds packets to the qdisc of the Ethernet interface. While processing and executing the qdisc is initially handled by Core 0, once the qdisc's quota is exhausted, the `__netif_schedule` function (cf. Figure 1) is invoked to schedule a TX SoftIRQ. This SoftIRQ, which is then executed by Core 1, is responsible for dequeuing packets from the qdisc via the `net_tx_action` function. Therefore, some of the load of W2E packet processing is offloaded to Core 1. These discussions also justify why Figures 7(a) and 8(a) show a considerable number of TX SoftIRQs for the W2E path, especially for the maximum throughput levels.

As mentioned above, the use of qdisc is triggered by the unavailability of the egress interface's TX queue. Since the results presented thus far are based on UDP traffic, we now turn our attention to TCP, which inherently adjusts its transmission rate to avoid packet drops caused by queue overflows. Our goal is to demonstrate that Observation 6 is also applicable to TCP traffic. For both APW6 and APW7, Table II shows the normalized number of processor cycles consumed by various stages of packet switching after the IP stack processing step (cf. Figure 1).[3] In this table, the 'Enqueue qdisc' column refers to the processing cycles consumed for queuing packets to the qdisc of the Ethernet interface.[4] The 'Direct Xmit' column indicates the processing cycles consumed by operations performed on packets that bypass the qdisc and are directly added to the egress NIC's driver.[5] The 'qdisc Xmit' column indicates the processor cycles consumed for operations on packets that are dequeued and transmitted from the qdisc.[6] The two sub-columns, 'Core 0' and 'Core 1' refer to the cores handling qdisc Xmit. Note that when using TCP flow, the measurements presented in Table II pertain to only data packets, not TCP ACK packets.

As Table II shows, when data packets are transmitted through the E2W path, no processor cycles are consumed by the Enqueue qdisc and qdisc Xmit stages. In contrast, these two processing stages do consume processor cycles when data packets are transmitted through the W2E path. Additionally, we observe that even though TCP employs congestion control to prevent packet loss due to AP's buffer overload, switching TCP data packets over the W2E path still benefits from dual-core processing.

Some of the implications of the discussions presented in this section are as follows. (i) For packets added to qdisc, both the execution of the qdisc and the addition of packets to the egress driver's TX buffer are performed by Core 1. Therefore, in scenarios where additional packet processing is required and should be offloaded to Core 1, such processing must be performed just before the addition to the TX buffer. To this end, an option is to invoke an eBPF program during the dequeue process of qdisc. Typical examples of such programs include dynamically dropping non-priority packets when the system detects congestion, modifying the packet's DSCP field to prioritize latency-sensitive flows at the final stage before transmission, or selectively redirect sensitive traffic to a monitoring system for inspection. (ii) The lack of a qdisc on the Wi-Fi interface, combined with the higher overhead of Wi-Fi egress processing (as demonstrated by our results in Observations 7, 8 and 9), can prevent the core assigned to the Ethernet interface from achieving maximum throughput over the E2W path. We observe this effect as the lower throughput of the E2W path compared to the W2E path on the APW6 platform. This problem becomes even more pronounced when additional packet processing is required along the E2W path. One potential solution to alleviate this bottleneck is the use of Receive Packet Steering (RPS); however, RPS can degrade cache efficiency [20], [39]. More critically, RPS increases contention for locks within the mac80211 module during packet enqueuing. To mitigate these issues, the number of RPS cores should be kept as low as possible (while ensuring that the cores are not overwhelmed), and lightweight packet batching mechanisms should be employed before packets are enqueued into the mac80211 module to reduce lock acquisition frequency.

TABLE II
NORMALIZED PROCESSOR CYCLES WITH DUAL-CORE CONFIGURATION

Results collected from APW6

	Enqueue qdisc	Direct Xmit	qdisc Xmit	
			Core 0	Core 1
E2W-UDP (852 Mbps)	0	1174616	0	0
E2W-TCP (849 Mbps)	0	1156818	0	0
W2E-UDP (949 Mbps)	16027	2704	47633	112503
W2E-TCP (904 Mbps)	10500	7098	46244	118097

Results collected from APW7

	Enqueue qdisc	Direct Xmit	qdisc Xmit	
			Core 0	Core 1
E2W-UDP (950 Mbps)	0	213041	0	0
E2W-TCP (941 Mbps)	0	185739	0	0
W2E-UDP (950 Mbps)	55899	5358	25202	72743
W2E-TCP (921 Mbps)	49345	6840	23962	64058

Observation 7: *For both E2W and W2E paths, the dual-core configuration consumes more power than the single-core configuration.*

Using APW6, in sub-figure (d) of Figures 5, 6, 7, and 8, we observe the following: (i) For both E2W and W2E paths, the power consumption of the dual-core configuration is higher than that of the single-core configuration, and (ii) the power consumption of the E2W path is higher than that of W2E. To

[3]We used the perf utility to perform these measurements. It is worth noting that using perf imposes some overhead, thereby, for instance, for the E2W path, the maximum achievable throughput when using a UDP flow is lower than the numbers reported before.

[4]These operations start with function `dev_qdisc_enqueue`.

[5]These operations start with function `dev_hard_start_xmit` when the function is called right after IP stack processing (i.e., bypassing qdisc).

[6]These operations begin with the function `dev_hard_start_xmit` when it is called by `__qdisc_run`.

better illustrate these differences, Tables III and IV summarize the normalized power consumption, computed as milliwatts (mW) per 1 Mbps of throughput, across three throughput levels for APW6 and APW7, respectively.

TABLE III
NORMALIZED POWER CONSUMPTION VALUES OF APW6

	100 Mbps	500 Mbps	Max
E2W Single-Core (mW)	3.410	1.426	1.108 (750 Mbps)
E2W Dual-Core (mW)	3.780	1.514	1.144 (893 Mbps)
% Dual-Core vs. Single-Core	10.85%	6.17%	3.21%
W2E Single-Core (mW)	3.140	1.142	0.953 (850 Mbps)
W2E Dual-Core (mW)	3.370	1.244	1.095 (949 Mbps)
% Dual-Core vs. Single-Core	7.32%	8.93%	8.60%
% E2W vs. W2E for Single-Core	8.60%	24.86%	16.26%
% E2W vs. W2E for Dual-Core	12.16%	21.70%	10.53%

TABLE IV
NORMALIZED POWER CONSUMPTION VALUES OF APW7

	100 Mbps	500 Mbps	Max
E2W Single-Core (mW)	5.017	1.120	0.631 (925 Mbps)
E2W Dual-Core (mW)	5.044	1.135	0.653 (950 Mbps)
% Dual-Core vs. Single-Core	0.52%	1.34%	3.33%
W2E Single-Core (mW)	5.013	1.044	0.564 (950 Mbps)
W2E Dual-Core (mW)	5.037	1.049	0.564 (950 Mbps)
% Dual-Core vs. Single-Core	0.48%	0.48%	0.36%
% E2W vs. W2E for Single-Core	0.08%	7.28%	12.08%
% E2W vs. W2E for Dual-Core	0.14%	8.20%	15.40%

TABLE V
NORMALIZED PROCESSOR CYCLES OF PACKET SWITCHING CORES ON APW6

	100 Mbps	500 Mbps	Max
E2W Single-Core	0.195	0.139	0.121 (750 Mbps)
E2W Dual-Core	0.207	0.148	0.126 (893 Mbps)
% Dual-Core vs. Single-Core	6.23%	6.24%	4.40%
W2E Single-Core	0.128	0.109	0.101 (850 Mbps)
W2E Dual-Core	0.144	0.129	0.124 (949 Mbps)
% Dual-Core vs. Single-Core	12.8%	18.1%	22.7%
% E2Wvs.W2E of Single-Core	52.4%	27.90%	19.21%
% E2Wvs.W2E of Dual-Core	43.56%	15.07%	1.39%

TABLE VI
NORMALIZED NUMBER OF CACHE INVALIDATIONS FOR PACKET SWITCHING CORES ON APW6

	100 Mbps	500 Mbps	Max
E2W Single-Core	7.16	1.46	1.04 (750 Mbps)
E2W Dual-Core	290.2867	147.55	89.34 (893 Mbps)
W2E Single-Core	6.91	1.39	0.89 (850 Mbps)
W2E Dual-Core	1253.5	694.14	476.65 (949 Mbps)

To justify the increase in power consumption observed in the dual-core configuration, we present the normalized processor cycles consumed by the packet-switching cores of APW6 in Table V. The results for APW7 are omitted due to space limitations. By comparing Tables III and V, we observe a clear numerical relationship between processor cycles and power consumption. For both AP platforms, we identify three key factors that contribute to the higher normalized processor cycles and, consequently, the increased normalized power consumption in the dual-core configuration. First, when a single core handles all packet processing tasks from ingress to egress, the RX buffer is checked less frequently compared to the case where a dedicated core handles ingress processing. As a result, the likelihood of packet accumulation increases, leading to more packets being processed per RX SoftIRQ. The higher packet aggregation, in turn, reduces the frequency of enabling and processing IRQs. This behavior can be observed in Figures 5, 6, 7, and 8, where the number of RX IRQs and RX SoftIRQs is lower in the single-core configuration. For instance, on the E2W path, the number of IRQs in the dual-core configuration increases by 6% and 28.8% compared to the single-core configuration at the 100 Mbps and 500 Mbps throughput levels, respectively. Similarly, the number of RX SoftIRQs increases by 10.9% and 50.4% at those same throughput levels. Second, we observe that the dual-core configuration results in a higher number of Layer-1 Data Cache (L1-DCache) invalidations. Table VI presents the normalized number of cache invalidations on APW6, calculated by dividing the total number of cache invalidations by the corresponding throughput level. The highlighted rows in the table reveal a higher cache invalidation rate for the dual-core configurations. Since each core has a dedicated L1 cache (this applies to both APW6 and APW7), in the dual-core configuration, packets stored in Core 1's L1 cache must be transferred to Core 0's L1 cache during the packet switching process. This inter-core data transfer results in a higher number of cache invalidations. It is worth noting that for the E2W path, even though Core 1 handles both ingress and egress processing, Core 0 handles TX IRQs for packet buffer claiming; therefore, Core 0 requires access to the buffer space of transmitted packets. Third, when two cores handle packet-switching processes, the number of context switches between kernel tasks increases. Using APW6, for the E2W path and 500 Mbps throughput, we measured these values for the single-core and dual-core configurations as 3361 and 5824, respectively. For the W2E path, these numbers are 3351 and 4369, respectively. Overall, our quantitative analyses align with the factors identified in prior works, such as [20], [40]–[42]. However, a detailed analysis of the individual impact of packet aggregation, cache behavior, and context switching on power consumption is left as future work. Another observation from Tables III, IV, V, and VI is that power consumption, normalized processor cycles, and normalized cache invalidations all decrease as throughput increases. In general, as network throughput increases, the number of processor cycles required to process each megabit of traffic typically decreases. This efficiency gain is attributed to the amortization of fixed per-packet overheads (e.g., interrupt handling, buffer allocation, and context switching) across a larger data volume. Higher throughput also improves batching efficiency (e.g., via NAPI), enhances cache locality, and reduces idle time, enabling more efficient processor utilization during packet processing.

TABLE VII
DEFINITION OF THE STAGES IN PACKET SWITCHING

Stage	**Description** (cf. Figure 1 for more information)
Total Cycles	The number of processor cycles per second consumed by the cores assigned to packet processing. This includes packet switching and other kernel operations.
Ingress Packet Processing	
Init RX-IRQ	The operations between the detection of an RX IRQ and the start of running an RX SoftIRQ.
Init Def-Work	The operations between scheduling a `ksoftirqd` thread or a tasklet and the start of deferred work.
RX SoftIRQ	The execution of an RX SoftIRQ until the start of running the driver's poll function. This includes the functions `__do_softirq` and `net_rx_action`, excluding the actual execution of driver's poll function.
Poll Function	The execution of the driver's poll function, which includes fetching packets from the device's RX buffer, performing necessary processing and passing the packets to the IP stack.
IP Stack Processing	
IP Stack	The execution of IP stack for processing incoming packets. This includes the operations starting with `ip_rcv` and before `__dev_queue_xmit`.
Egress Packet Processing	
Init Xmit	The execution of the functions responsible for pushing packets into the `qdisc` or the driver's TX buffer. This includes the operations started by the `__dev_queue_xmit` function. This also includes the execution of `qdisc_run` on the core handling ingress packet processing.
Init TX-IRQ	The operations between the detection of a TX IRQ and the start of running a TX SoftIRQ.
TX SoftIRQ	The execution of a SoftIRQ until the start of the TX Reclaim stage or the TX Qdisc stage.
TX Qdisc	The execution of `qdisc_run` on the core assigned to Ethernet NIC's IRQ. This stage includes the transfer of packets from the `qdisc` to driver's TX buffer and packet transmission from this buffer.
TX Reclaim	The process of reclaiming transmission resources (SKBs) after packets have been sent out.

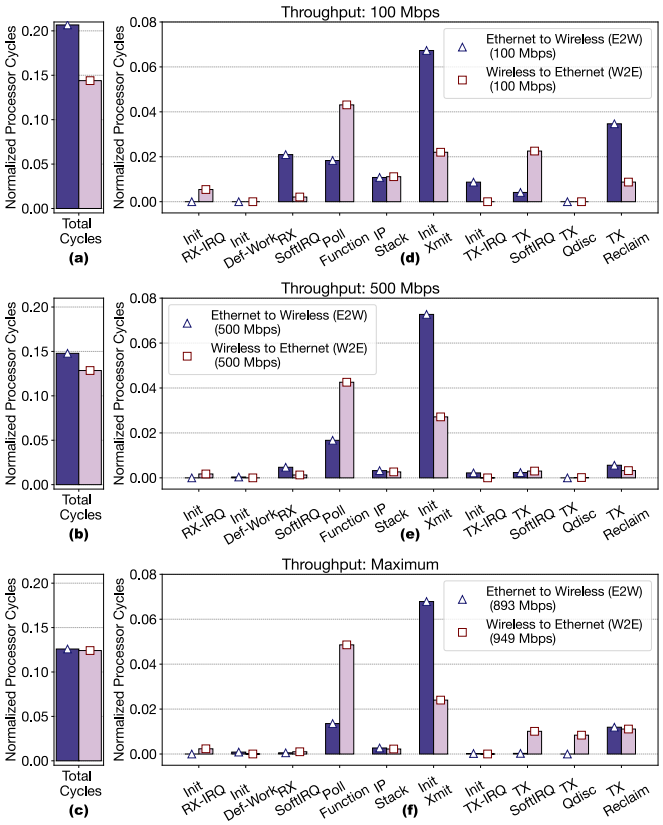

Fig. 9. Microanalysis results from the APW6 platform. (a), (b), and (c): The normalized cycles consumed by all cores for the E2W (triangle) and W2E (square) paths. (d), (e), and (f): The x-axis corresponds to the various stages outlined in Table VII, providing a detailed breakdown of processing costs. For each packet switching stage, the values represent the normalized number of cycles consumed by that stage alone. All results are for the dual-core configuration.

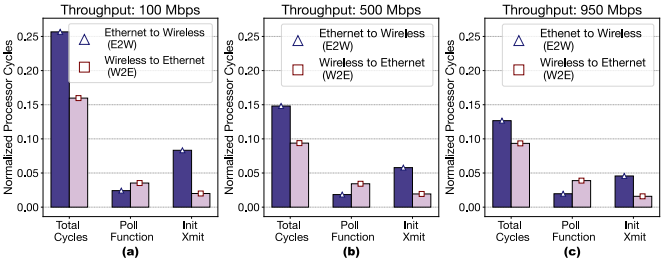

Fig. 10. Microanalysis results from the APW7 platform. The x-axis corresponds to the various stages outlined in Table VII. All results are for the dual-core configuration.

V. ANALYZING THE STAGES OF PACKET SWITCHING: A MICROANALYSIS APPROACH

The metrics and analyses presented in the previous section enabled us to describe the overarching operational framework and discern the differences in packet switching considering path and core assignment configuration. To deepen our understanding of these operations and the root causes of the observed differences, this section adopts a more granular approach by conducting a micro-analysis of packet switching stages. Our methodology is structured as follows. We decompose packet switching operations into distinct stages, as Table VII illustrates. For APW6 and APW7, Figures 9 and 10 present the normalized total number of core cycles and the cycles consumed per stage of packet switching. Due to space limitations, however, Figure 10 only shows the normalized number of cycles attributed to two important stages, which are discussed in more detail in this section. We used the same normalization method explained in Observation 3. To quantify the processing load associated with each stage, we measured the number of processor cycles using the perf tool. The details of the mappings and the corresponding codes are available in the GitHub repository of this paper [32].

Observation 8: *The two most process-intensive stages are the Poll Function stage and the Init Xmit stage. For the WiFi interface in particular, the processing load of these stages is higher than that of the Ethernet interface. This is a general observation for APs based on the SoftMAC class of WiFi devices.*

The Poll Function stage of WiFi ingress processing incurs higher overhead than that of the Ethernet interface. For APW6, Figures 9(d), (e), and (f) show that at 100 Mbps, 500 Mbps, and maximum throughput levels, the number of cycles con-

sumed by the Poll Function stage in the W2E path is 2.3×, 2.5×, and 3.8× higher, respectively, than in the E2W path. Similarly, for APW7, Figures 10(a), (b), and (c) show that at 100 Mbps, 500 Mbps, and maximum throughput levels, the number of cycles consumed by the Poll Function stage in the W2E path is 1.5×, 1.9×, and 2.0× higher, respectively, than in the E2W path.

After retrieving the incoming packets from the RX buffer, the Ethernet driver performs necessary validations (e.g., error checking, packet size validation) and passes the packets to the IP stack for further processing. In contrast, the WiFi driver performs more complex tasks and needs to process 802.11 headers, which are more intricate than Ethernet headers. This involves extracting and processing additional fields such as QoS control, sequence control, and security-related fields. After parsing the headers of MAC Protocol Data Units (MPDU), the driver performs validations such as error checking and packet size validation on each MPDU. If an MPDU is part of a fragmented frame sequence, the driver handles the fragmentation and reassembly of packets. After these parsing and validation steps, the driver strips off the 802.11 headers and additional encapsulation headers before handing the packet to the IP layer.

The Init Xmit stage of the WiFi egress processing incurs higher overhead than that of the Ethernet interface. For APW6, Figure 9 shows that for the 100, 500 Mbps and maximum throughput levels, the processor cycles consumed by the Init Xmit stage of the E2W path are 3x, 2.7x and 2.6x higher, respectively, than the Init Xmit stage of the W2E path. For APW7, Figure 10 shows that for the 100, 500 Mbps and maximum throughput levels, the processor cycles consumed by the Init Xmit stage of the E2W path are 4.2x, 3.0x and 2.8x higher, respectively, than the Init Xmit stage of the W2E path. Recall that the Init Xmit stage starts after the completion of IP stack processing. At this point, the packet is passed to the netdev subsystem of the egress NIC. For the E2W path, since there is no qdisc (cf. Section IV, Observation 6), packets are passed to the mac80211 module. In addition to tasks such as frame aggregation (e.g., A-MPDU), this module implements complex queuing methods to establish airtime and QoS-aware delivery of egress traffic. More specifically, the mac80211 module strives to allocate a fair share of airtime to each station, and the priority of the traffic (i.e., voice, video, best-effort, and background) also affects the order of packet transmissions [3], [5]. These operations introduce queue management and locking overheads that increase with throughput. Below the mac80211 module, the WiFi driver (iwlwifi or mt7996) implements tasks such as retransmissions, rate control, power management, medium access control, encryption, and QoS. All of these factors contribute to the higher processing load of egress processing compared to ingress processing. Our deeper analysis of the E2W path revealed that mac80211 processing accounts for, on average, 62.6% of the Init Xmit sub-stage on APW6 and 57.1% on APW7.

For the W2E path, the tasks for the Init Xmit stage are fewer and less complex. Specifically, for packets that are directly transmitted, the processing load of the Ethernet drivers (bcmgenet or mtk_eth_soc) is lower than that of the mac80211 module and the WiFi driver (iwlwifi or mt7996). This is because the Ethernet drivers do not perform complex tasks such as QoS-based queuing and airtime-aware channel access, which are required by WiFi drivers. However, on the W2E path, as the throughput increases, the number of packets that cannot bypass the qdisc increases, and the overhead of the qdisc also rises. We elaborate on the processing load of the qdisc in Observation 9.

The presented discussions focused on a SoftMAC-based WiFi NIC utilizing the mac80211 module. In contrast, FullMAC WiFi NICs have the potential to significantly reduce packet processing overhead for ingress and egress on the WiFi interface. However, this reduction in overhead comes at the expense of programmability and monitoring capabilities, as most queuing and MAC functionalities are offloaded to the NIC. Exploring these trade-offs remains a direction for future research.

Observation 9: *The W2E path is more efficient than the E2W path. However, as throughput increases, the difference in processing overhead between these two paths reduces.*

For APW6, comparing Figures 9(a), (b), and (c), we observe that across all throughput levels, the total number of processor cycles consumed by the W2E path is lower than that of the E2W path. Similarly, for APW7, Figure 10 shows the same trend. In terms of power consumption, Tables III and IV show higher power consumption for the E2W path compared to the W2E path for both APW6 and APW7. These trends align with our previous discussions. For instance, in Observation 8, we explained the higher processing demand of WiFi egress processing compared to ingress processing for WiFi interfaces based on the SoftMAC architecture.

For both APW6 and APW7, from Figures 9 and 10 we also observe that the efficiency of both paths improves as throughput increases. As discussed in Observation 7, this improvement occurs because a higher number of packets are processed per stage, which enhances efficiency in most stages (e.g., RX SoftIRQ, IP Stack). However, for some stages, such as TX Reclaim, we observe a different trend: as throughput increases from 100 Mbps to 500 Mbps and then to the maximum value, efficiency initially improves but subsequently declines. At the 100 Mbps throughput level, fixed costs (e.g., memory allocation) dominate processing costs, leading to lower efficiency. As throughput increases, these fixed costs are amortized over more packets, improving efficiency. However, at the highest throughput, the working set of data may exceed the processor's cache capacity, resulting in frequent cache evictions and increased memory latency [20]. Additionally, high traffic volumes can create contention for shared resources, such as spinlocks in the qdisc or the driver's TX buffer. These factors contribute to the observed decline in efficiency, which we discuss in greater detail in the following paragraphs.

Another important observation from Figures 9 and 10 is that as throughput increases, the W2E path exhibits a slower decline in the total number of processor cycles used, compared to the E2W path. This behavior is related to the processing overhead associated with scheduling and running the qdisc for the Ethernet interface. As discussed in Observation 6,

the dual-core configuration leverages both cores in the W2E path. By adding packets to the qdisc, egress processing is offloaded to the core assigned to the Ethernet NIC (Core 1), effectively distributing the packet switching load across the two cores. However, the processing load of the egress side in the W2E path increases as more packets are added to the qdisc instead of being transmitted directly. Adding a qdisc to the packet switching path introduces additional overhead due to the extra processing and synchronization required to manage its operations. When packets pass through a qdisc, they must undergo enqueue and dequeue operations, which involve inserting and removing packets from internal data structures, such as linked lists. These processes require memory access and often necessitate locking mechanisms to ensure consistency. Moreover, the dequeueing process involves applying scheduling logic to determine the transmission order, further increasing complexity and processing cost. In contrast, bypassing the qdisc and sending packets directly to the driver eliminates much of this complexity. However, as throughput increases, so does the number of packets that cannot bypass the qdisc. Further analysis (not included in this paper) revealed that, at the maximum throughput level of the W2E path, the percentage of packets added to the qdisc (rather than being directly transmitted) is 76% for APW6 and 92% for APW7. The increasing processing demand of the qdisc is reflected in the higher number of processor cycles consumed by the TX SoftIRQ and TX Qdisc stages. For the W2E path on APW6, increasing the throughput from 500 Mbps to the maximum throughput results in a $3\times$ and $61\times$ increase in the number of processor cycles consumed by these two sub-stages, respectively. Similarly, for APW7, increasing the throughput from 500 Mbps to the maximum throughput results in a $63\times$ and $128\times$ increase in the number of processor cycles consumed by the same two sub-stages, respectively.

The discussion presented in this observation, along with Observation 6, highlights an important implication. Although Observation 6 shows that packet switching in the W2E path can benefit from distributing the processing load over two cores, the use of qdisc incurs a higher packet processing cost compared to directly adding packets to the driver's TX buffer on the egress interface. Therefore, in scenarios with traffic flows that do not require advanced scheduling, prioritization, or shaping, the system could bypass the qdisc, for example, by using the 'redirect' functionality of eXpress Data Path (XDP), to reduce processor cycles, overhead, and energy consumption. In such cases, a larger TX buffer may be needed to absorb variations between ingress and egress transmission rates. Additionally, depending on application demands, simple queue management mechanisms could be implemented within the device driver, directly before the TX buffer.

VI. Related Work

To the best of our knowledge, there is no study on understanding and enhancing the NIC-to-NIC data plane of WiFi APs. The existing works related to packet switching on WiFi APs are primarily focused on packet scheduling, queuing, and airtime fairness [3], [5], [16], [43]–[47]. For instance, the increased latency and jitter due to excessive queueing (bufferbloat) is discussed in [3]. In another category of related works, tools have been proposed in [4] and [48] to facilitate event tracing and provide additional real-time insights into the operation of the WiFi stack. Such tools can be used in addition to the statistics provided by the kernel to better understand and enhance the kernel's data path. It is also worth noting that this paper, as well as a large body of existing works, such as [3]–[5], [16], [43]–[50], all use a SoftMAC-based WiFi driver architecture. Sample commercial platforms based on this architecture include Qualcomm's `ath9k`, `ath10k`, and `ath11k`, Intel's iwlwifi, and MediaTek's `mt76`.

There exist several works on the evaluation and enhancement of software packet switching in *high-performance settings* utilizing servers with Xeon (x86-64) processors. The study [41] demonstrated that binding all forwarding operations of each packet to a single core reduces cache invalidation rate and spinlock contention compared to multi-core packet switching. A similar observation has been reported in [23]. Towards achieving predictable packet switching performance, [42] shows that cache contention is the leading cause of performance variations. They also examined the potential benefits of contention-aware task scheduling and found that it provides minimal performance improvement. Dynamic configurations based on workload demands are emphasized in [21]. Settings such as disabling Hyper-Threading (specific to x86-64 processors) and processor core isolation are shown to be crucial for achieving high and stable performance. The study [51] shows that switching more than one million flows results in the considerably higher overhead of IP routing, which occurs when the routing table cannot fit entirely into the processor's cache. The disparity between processor utilization and the number of cycles consumed has been mentioned in [22]; however, they did not discuss the underlying causes of the problem.

Various works have modified the kernel path [52], [53] or proposed novel algorithms and methods for the dynamic configuration of packet-switching systems [54], [55]. To minimize delays for high-priority data flows, [54] proposes a method that involves assigning incoming packets to different RX buffers based on their priority levels and then employing various scheduling techniques to efficiently manage the resources assigned to processing incoming packets. The problem of finding optimal configuration parameters for a software switch to achieve maximum throughput and minimum latency has been addressed in [55]. Kafe [52] enhances kernel packet switching by introducing a cache-optimized SKB allocator that recycles pre-allocated buffers efficiently. This method reduces the overhead associated with frequent allocation and de-allocation of memory buffers, lowering cache misses. The authors in [53] proposed ZygOS, a system designed to achieve low tail latency in microsecond-scale networked tasks. ZygOS incorporates advanced interrupt handling and scheduling techniques to enhance performance. Given the flexibility of software switches in managing physical resources such as the number of processor cores and capacity of RX buffers, [30] propose a model to instantaneously estimate the minimum number of processor cores required to meet given QoS criteria.

In this paper, we leveraged several best practices, including IRQ affinity and core isolation, to enhance switching performance. Additional optimizations are orthogonal to our work and can be employed for further performance enhancements. For instance, dynamic IRQ affinity assignment and multi-queue NICs can be utilized to allocate resources based on the system's load. Furthermore, studying the impact of the number of flows and complex Network Function Virtualization (NFV) tasks (e.g., firewall, encryption, monitoring) on cache performance remains an area for future research.

Given the importance of communications between NIC and *user-space applications* and Virtual Machines (VMs) for cloud computing and NFV environments, understanding and enhancing this communication has also received attention. The study presented in [41] demonstrated that, under conditions of high traffic, a single core dedicated to packet switching and user-space processing allocates less than 2% of its resources to user-space tasks. Focused on packet switching between kernel and user-space, [20] demonstrated that more than 50% of the overhead is caused by packet copying operations. The overhead of traffic switching between the user-space and two NICs supporting 802.11ad and 802.11ac has been studied in [56]. Their utilized platform has four powerful cores and four small cores, and when the IRQs are assigned to small cores, sub-optimal throughput is observed. The authors of [57] revealed that packet exchanges between NIC and user-space applications do not accurately account for the application's processor demand. They proposed a kernel modification of the SoftIRQ processing to address this problem. In [50], the authors provide applications with real-time access to MAC primitives, enabling direct control over packet transmissions and retransmissions at the user space level.

As WiFi APs continue to be deployed in a wide range of applications, it is anticipated that an increasing number of NFV instances will run on these APs [7]–[9], [14], [58]–[61]. For example, implementing security functions like firewalls and intrusion detection directly on APs can provide faster threat responses, while deploying content caching applications at the edge can significantly reduce bandwidth consumption and improve user experience by delivering content more quickly. Therefore, an area of future work is to study the implications of packet switching performance for running NFV in the kernel or user-space and to adopt various optimization methods [29], [62], [63].

Existing works on user-space packet switching (i.e., kernel bypass methods) demonstrate superior performance compared to Linux's layer-2 and layer-3 switching [21], [22], [30]. In [22], the authors showed that for packet switching between two NICs, the performance of the Linux bridge, Linux's IP forwarding, OVS-kernel and OVS-DPDK are 1.11, 1.58, 1.88 and 11.31 million packet per second, respectively. Implementation and analysis of user-space packet switching on WiFi APs is left as a future work.

VII. Conclusion

As WiFi APs continue to evolve toward higher performance and broader functional integration, understanding the behavior and bottlenecks of Linux's software-based packet switching becomes increasingly important. This paper systematically examined the packet switching paths in Linux-based APs, highlighting the validity domains of statistics, how processing is distributed across cores, and how architectural choices (e.g., SoftMAC, qdisc) influence performance. These observations challenge the reliability of conventional kernel statistics and reveal nuanced system behaviors that influence both the interpretation of these statistics and overall performance. For instance, our findings show that widely used system monitoring tools can significantly misrepresent processor load due to hidden costs such as interrupt-context SoftIRQ execution. We also observe that qdisc introduces both benefits (e.g., parallelism) and costs (e.g., overhead), and that asymmetries in the W2E and E2W paths result in different core utilization patterns. Together, these findings offer new insights into monitoring accuracy, core assignment, and interrupt and deferred network task handling, which are items that have not been previously emphasized in the context of WiFi APs.

While this study primarily focuses on statistics provided by standard Linux monitoring tools (such as the `/proc` file system and perf), additional insights into packet switching operations could be gained through the use of eBPF, which enables dynamic, real-time instrumentation and observation of kernel activity. Developing integrated toolchains that correlate low-level, fine-grained data (e.g., collected by perf or eBPF) with metrics reported by high-level, system-wide tools (e.g., mpstat) could help close the visibility gap, enabling more accurate analysis and automated system tuning. Enhancing observability of these internal behaviors brings us closer to building APs that are not only performant, but also observable and adaptive to application needs.

It is also important to note that while we focused on packet switching on APs, the findings apply to any Linux-based host. For example, our results indicate that relying solely on the reported processor utilization attributed to SoftIRQs may not be an effective method for detecting IRQ overload or work performed within the IRQ context, such as that which can occur during Denial-of-Service (DoS) attacks on IoT devices [64]. Because these behaviors stem from the Linux kernel's networking and interrupt handling subsystems, the implications extend beyond APs to other Linux-based systems such as IoT gateways and devices. Nonetheless, the extent of these effects may vary depending on the kernel version, device drivers, and NIC architecture. Future research may also explore dynamic system configurations, such as IRQ affinity assignment or selectively bypassing qdisc, based on real-time load characteristics. In addition, techniques such as cache-aware NFV, user-space packet switching, multi-queue drivers with multiple RX and TX buffers, and refined RPS strategies offer promising directions to further enhance APs performance.

Acknowledgment

This work was supported by NSF grant #2138633 and an internal grant from Santa Clara University's Center for Sustainability.

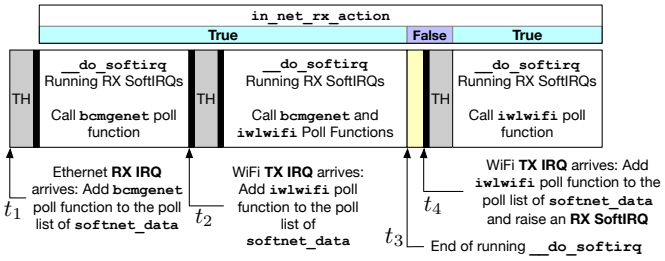

Fig. 11. At time instances t_2 and t_4, the WiFi NIC generates TX IRQs. In both cases, after adding the poll function of `iwlwifi` to the poll list of `softnet_data`, the RX SoftIRQ is used to call this poll function, which decides if reception, transmission, or both types of packet handling must be performed.

APPENDIX
RX SOFTIRQ EXECUTION ON EGRESS SIDE

To visualize and better understand why RX SoftIRQs are triggered by the egress NIC, we analyze the code section that determines which SoftIRQ type must be scheduled. We explain the kernel's operation by presenting a single-core scenario in Figure 11, focusing on the APW6 platform. At time t_1, an Ethernet RX IRQ arrives, the TH processing of this IRQ uses the function `__napi_schedule` to add the poll function of the driver to the poll list of the `softnet_data` structure of the core. When the `__do_softirq` activates an RX SoftIRQ to call the poll function, the `in_net_rx_action` variable is set to `true`. At time t_2, while an RX SoftIRQ is running the poll function of bcmgenet driver, the WiFi NIC (egress) generates a TX IRQ. Again, the function `__napi_schedule` is used to add the poll function of the WiFi NIC to the `softnet_data` structure of this core. At this time, since an RX SoftIRQ is in progress, the `in_net_rx_action` variable of the `softnet_data` structure on this core is `true`; thereby, there is no need to schedule a new SoftIRQ instance. In other words, the RX SoftIRQ instance that is already running will call the poll function of the iwlwifi driver.

At time t_3, there is no ongoing RX SoftIRQ and the value of `in_net_rx_action` is `false`. When the WiFi NIC generates an IRQ at time t_4, within the TH processing, the `__napi_schedule` function calls function `__raise_softirq_irqoff` (cf. Figure 1) to mark an RX SoftIRQ as pending, ensuring that it will be processed at the next opportunity when the kernel checks for pending SoftIRQs. Notably, when calling function `__raise_softirq_irqoff` by `__napi_schedule`, the kernel always uses RX SoftIRQ type as the argument of the function. Therefore, considering the zero number of TX SoftIRQs in Figures 5(a) and 6(a), this phenomenon occurs because TX IRQs are handled by RX SoftIRQs.

REFERENCES

[1] E. Reshef and C. Cordeiro, "Future directions for Wi-Fi 8 and beyond," *IEEE Communications Magazine*, vol. 60, no. 10, pp. 50–55, 2022.

[2] F. M. Intelligence, "Gigabit Wi-Fi Access Point Market Outlook for 2024 to 2034." FMI, 2022. [Online]. Available: https://www.futuremarketinsights.com/reports/gigabit-wi-fi-access-point-market

[3] T. Høiland-Jørgensen, M. Kazior, D. Täht, P. Hurtig, and A. Brunstrom, "Ending the Anomaly: Achieving Low Latency and Airtime Fairness in WiFi," in *USENIX Annual Technical Conference*, 2017, pp. 139–151.

[4] J. Sheth and B. Dezfouli, "Monfi: A tool for high-rate, efficient, and programmable monitoring of wifi devices," in *IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2021, pp. 1–7.

[5] J. Sheth, C. Miremadi, A. Dezfouli, and B. Dezfouli, "EAPS: Edge-Assisted Predictive Sleep Scheduling for 802.11 IoT Stations," *IEEE Systems Journal*, vol. 16, no. 1, pp. 591–602, 2022.

[6] Wireless Broadband Alliance, "WBA Annual Industry Report 2025," https://wballiance.com/wba-annual-industry-report-2025-shows-steep-growth-in-confidence-for-wi-fi-and-openroaming/, 2025, accessed: 2025-04-26.

[7] J. Schulz-Zander, C. Mayer, B. Ciobotaru, R. Lisicki, S. Schmid, and A. Feldmann, "Unified programmability of virtualized network functions and software-defined wireless networks," *IEEE Transactions on Network and Service Management (TNSM)*, vol. 14, no. 4, pp. 1046–1060, 2017.

[8] M. S. Bonfim, K. L. Dias, and S. F. Fernandes, "Integrated NFV/SDN architectures: A systematic literature review," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–39, 2019.

[9] C. Blumschein, I. Behnke, L. Thamsen, and O. Kao, "Differentiating Network Flows for Priority-Aware Scheduling of Incoming Packets in Real-Time IoT Systems," in *IEEE 25th International Symposium On Real-Time Distributed Computing (ISORC)*, 2022, pp. 1–8.

[10] "IPQ8074," n.d., accessed: 2024-04-05. [Online]. Available: https://www.qualcomm.com/products/internet-of-things/networking/wi-fi-networks/ipq8074

[11] S.-Y. Wang and J.-C. Chang, "Design and implementation of an intrusion detection system by using extended BPF in the Linux kernel," *Journal of Network and Computer Applications*, vol. 198, p. 103283, 2022.

[12] Q. Ren, L. Zhou, Z. Xu, Y. Zhang, and L. Zhang, "PacketUsher: Exploiting DPDK to accelerate compute-intensive packet processing," *Computer Communications*, vol. 161, pp. 324–333, 2020.

[13] Cisco Systems, "Application hosting on catalyst access points," 2023, White Paper, Accessed: 2025-04-26. [Online]. Available: https://www.cisco.com/c/en/us/products/collateral/wireless/access-points/white-paper-c11-744304.html

[14] Wireless Broadband Alliance, "WBA Annual Industry Report 2021," https://wballiance.com/resource/wba-annual-industry-report-2021/, 2021, accessed: 2025-04-26.

[15] J. J. Aleixendri, A. Betzler, and D. Camps-Mur, "A practical approach to slicing Wi-Fi RANs in future 5G networks," in *2019 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2019, pp. 1–6.

[16] M. Richart, J. Baliosian, J. Serrat, J.-L. Gorricho, and R. Agüero, "Slicing with guaranteed quality of service in WiFi networks," *IEEE Transactions on Network and Service Management*, vol. 17, no. 3, pp. 1822–1837, 2020.

[17] ——, "Slicing in wifi networks through airtime-based resource allocation," *Journal of Network and Systems Management*, vol. 27, pp. 784–814, 2019.

[18] S. Zehl, A. Zubow, and A. Wolisz, "Hotspot slicer: Slicing virtualized home Wi-Fi networks for air-time guarantee and traffic isolation," in *IEEE 18th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. IEEE, 2017, pp. 1–3.

[19] J. Chen and B. Dezfouli, "Predictable bandwidth slicing with open vswitch," in *IEEE Global Communications Conference (GLOBECOM)*, 2021, pp. 1–6.

[20] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, "Understanding host network stack overheads," in *Proceedings of the ACM SIGCOMM*, 2021, pp. 65–77.

[21] V. Fang, T. Lévai, S. Han, S. Ratnasamy, B. Raghavan, and J. Sherry, "Evaluating software switches: Hard or hopeless?" EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2018-136, Oct 2018.

[22] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "Performance characteristics of virtual switching," in *IEEE 3rd International Conference on Cloud Networking (CloudNet)*, 2014, pp. 120–125.

[23] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: Exploiting parallelism to scale software routers," in *ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 15–28.

[24] P. Okelmann, L. Linguaglossa, F. Geyer, P. Emmerich, and G. Carle, "Adaptive batching for fast packet processing in software routers using machine learning," in *IEEE 7th International Conference on Network Softwarization (NetSoft)*. IEEE, 2021, pp. 206–210.

[25] A. Mestres, A. Rodriguez-Natal, J. Carner, P. Barlet-Ros, E. Alarcón, M. Solé, V. Muntés-Mulero, D. Meyer, S. Barkai, M. J. Hibbett *et al.*, "Knowledge-defined networking," *ACM SIGCOMM Computer Communication Review*, vol. 47, no. 3, pp. 2–10, 2017.

[26] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM workshop on hot topics in networks*, 2016, pp. 50–56.

[27] G. Bernárdez, J. Suárez-Varela, A. López, B. Wu, S. Xiao, X. Cheng, P. Barlet-Ros, and A. Cabellos-Aparicio, "Is machine learning ready for traffic engineering optimization?" in *IEEE 29th International Conference on Network Protocols (ICNP)*. IEEE, 2021, pp. 1–11.

[28] C. Powell, C. Desiniotis, and B. Dezfouli, "The fog development kit: A platform for the development and management of fog systems," *IEEE Internet of Things Journal*, vol. 7, no. 4, pp. 3198–3213, 2020.

[29] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, L. Iannone, and J. Roberts, "Comparing the performance of state-of-the-art software switches for NFV," in *CoNEXT*, 2019, pp. 68–81.

[30] G. A. Gallardo, B. Baynat, and T. Begin, "Performance modeling of virtual switching systems," in *IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2016, pp. 125–134.

[31] OpenWrt, "OpenWrt Platforms," 2024. [Online]. Available: https://openwrt.org/docs/platforms/start

[32] SIOTLAB, "Understanding and Enhancing Linux Kernel-based Packet Switching on WiFi Access Points," 2024. [Online]. Available: https://github.com/SIOTLAB/wifi-packet-switching-analysis

[33] T. Hoeiland-Joergensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet, "The FlowQueue-CoDel Packet Scheduler and Active Queue Management Algorithm," RFC 8290, Internet Engineering Task Force, January 2018. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8290

[34] Broadcom, "BCM5420: Single Port RGMII SGMII Gigabit Ethernet Transceiver," n.d., accessed: 2024-04-10. [Online]. Available: https://www.broadcom.com/products/ethernet-connectivity/phy-and-poe/copper/gigabit/bcm54210

[35] B. Dezfouli, I. Amirtharaj, and C.-C. Li, "EMPIOT: An energy measurement platform for wireless IoT devices," *Journal of Network and Computer Applications*, vol. 121, pp. 135–148, 2018.

[36] Linux Kernel, https://github.com/torvalds/linux/tree/master, accessed: 2025-04-10.

[37] The Linux Kernel, "ftrace - Function Tracer," n.d., accessed: 2024-02-10. [Online]. Available: https://www.kernel.org/doc/html/v6.0/trace/ftrace.html

[38] P. H. Isolani, M. Claeys, C. Donato, L. Z. Granville, and S. Latré, "A survey on the programmability of wireless mac protocols," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1064–1092, 2019.

[39] J. Lei, M. Munikar, K. Suo, H. Lu, and J. Rao, "Parallelizing packet processing in container overlay networks," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21, 2021, p. 261–276.

[40] H. Ghasemirahni, T. Barbette, G. P. Katsikas, A. Farshin, A. Roozbeh, M. Girondi, M. Chiesa, G. Q. Maguire Jr, and D. Kostić, "Packet order matters! improving application performance by deliberately delaying packets," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 807–827.

[41] R. Bolla and R. Bruschi, "PC-based software routers: High performance and application service support," in *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, 2008, pp. 27–32.

[42] M. Dobrescu, K. Argyraki, and S. Ratnasamy, "Toward predictable performance in software packet-processing platforms," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012, pp. 141–154.

[43] T. Høiland-Jørgensen, D. Täht, and J. Morton, "Piece of cake: a comprehensive queue management solution for home gateways," in *IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. IEEE, 2018, pp. 37–42.

[44] M. Richart, J. Baliosian, J. Serrati, J.-L. Gorricho, R. Agüero, and N. Agoulmine, "Resource allocation for network slicing in WiFi access points," in *13th International conference on network and service management (CNSM)*. IEEE, 2017, pp. 1–4.

[45] P. H. Isolani, N. Cardona, C. Donato, G. A. Pérez, J. M. Marquez-Barja, L. Z. Granville, and S. Latré, "Airtime-based resource allocation modeling for network slicing in IEEE 802.11 RANs," *IEEE Communications Letters*, vol. 24, no. 5, pp. 1077–1080, 2020.

[46] F. Canbal, Y. B. Ozgun, M. S. Kuran, G. Venkatesan, and N. Canpolat, "Wi-fi qos management program: Bridging the qos gap of multimedia traffic in wi-fi networks," *IEEE Communications Magazine*, 2023.

[47] S. Deng, X. Guan, Z. Sun, S. Zhao, T. Shen, X. Chen, T. Duan, Y. Wang, J. Pan, Y. Wu *et al.*, "Coorp: Satisfying low-latency and high-throughput requirements of wireless network for coordinated robotic learning," *IEEE Internet of Things Journal*, vol. 10, no. 3, pp. 1946–1960, 2022.

[48] J. Sheth, V. Ramanna, and B. Dezfouli, "Flip: A framework for leveraging ebpf to augment WiFi access points and investigate network performance," in *Proceedings of the 19th ACM International Symposium on Mobility Management and Wireless Access*, 2021, pp. 117–125.

[49] M. Vanhoef, X. Jiao, W. Liu, and I. Moerman, "Testing and improving the correctness of Wi-Fi frame injection," in *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2023, pp. 287–292.

[50] G. Cena, S. Scanzio, and A. Valenzano, "SDMAC: a software-defined MAC for Wi-Fi to ease implementation of soft real-time applications," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 6, pp. 3143–3154, 2018.

[51] D. Raumer, F. Wohlfart, D. Scholz, P. Emmerich, and G. Carle, "Performance exploration of software-based packet processing systems," *Leistungs-, Zuverlässigkeits-und Verlässlichkeitsbewertung von Kommunikationsnetzen und verteilten Systemen*, vol. 8, 2015.

[52] C.-H. Hong, K. Lee, J. Hwang, H. Park, and C. Yoo, "Kafe: Can os kernels forward packets fast enough for software routers?" *IEEE/ACM Transactions on Networking*, vol. 26, no. 6, pp. 2734–2747, 2018.

[53] G. Prekas, M. Kogias, and E. Bugnion, "Zygos: Achieving low tail latency for microsecond-scale networked tasks," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017, p. 325–341.

[54] T. Meyer, D. Raumer, F. Wohlfart, B. E. Wolfinger, and G. Carle, "Low latency packet processing in software routers," in *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*. IEEE, 2014, pp. 556–563.

[55] K. Suksomboon, N. Matsumoto, S. Okamoto, M. Hayashi, and Y. Ji, "Configuring a software router by the erlang-k-based packet latency prediction," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 422–437, 2018.

[56] I. Khan, M. Ghoshal, S. Aggarwal, D. Koutsonikolas, and J. Widmer, "Multipath tcp in smartphones equipped with millimeter wave radios," in *Proceedings of the 15th ACM Workshop on Wireless Network Testbeds, Experimental evaluation & Characterization*, 2022, pp. 54–60.

[57] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella, "Iron: Isolating network-based cpu in container environments," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 313–328.

[58] V. M. Martínez, M. R. Ribeiro, and V. F. Mota, "Wi-Fi faces the new wireless ecosystem: a critical review," *Annals of Telecommunications*, pp. 1–17, 2023.

[59] B. Wu, J. Zeng, L. Ge, S. Shao, Y. Tang, and X. Su, "Resource allocation optimization in the NFV-enabled MEC network based on game theory," in *IEEE International Conference on Communications (ICC)*. IEEE, 2019, pp. 1–7.

[60] S. Yang, F. Li, S. Trajanovski, R. Yahyapour, and X. Fu, "Recent advances of resource allocation in network function virtualization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 2, pp. 295–314, 2020.

[61] R. Riggio, T. Rasheed, and R. Narayanan, "Virtual network functions orchestration in enterprise WLANs," in *IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2015, pp. 1220–1225.

[62] R. Iyer, L. Pedrosa, A. Zaostrovnykh, S. Pirelli, K. Argyraki, and G. Candea, "Performance contracts for software network functions," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 517–530.

[63] F. Pereira, F. M. Ramos, and L. Pedrosa, "Automatic parallelization of software network functions," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024, pp. 1531–1550.

[64] B. Tushir, Y. Dalal, B. Dezfouli, and Y. Liu, "A quantitative study of ddos and e-ddos attacks on wifi smart home devices," *IEEE Internet of Things Journal*, 2020.