EAPS: Edge-Assisted Predictive Sleep Scheduling for 802.11 IoT Stations

Jaykumar Sheth*, Cyrus Miremadi*, Amir Dezfouli†, and Behnam Dezfouli*
*Internet of Things Research Lab, Department of Computer Science and Engineering, Santa Clara University, USA
{jsheth, cmiremadi, bdezfouli}@scu.edu
†Commonwealth Scientific and Industrial Research Organisation (CSIRO), Sydney, Australia
{amir.dezfouli@data61.csiro.au}

Abstract—The broad deployment of 802.11 (a.k.a., WiFi) access points and the significant energy-efficiency improvement of 802.11 transceivers have resulted in increasing interest in building 802.11-based IoT systems. Unfortunately, the power saving mechanisms of 802.11 fall short when used in IoT applications, especially because they do not take into account the delays caused by various factors such as buffering, interference, and round-trip delay. In this paper, we present *edge-assisted predictive sleep scheduling* (EAPS) to adjust the sleep duration of stations while they are expecting downlink packets. We first implement a Linux-based access point that enables us to collect parameters affecting communication latency. Using this access point, we build a testbed that, in addition to offering traffic pattern customization, replicates the characteristics of real-world environments. We then use multiple machine learning algorithms to predict downlink packet delivery. Our empirical evaluations confirm that with EAPS the energy consumption of IoT stations is as low as PSM, whereas the delay of packet delivery is close to the case where the station is always awake.

Index Terms—Energy Efficiency, Delay, Wireless Communication, Machine Learning, Edge Computing.

1 Introduction

Nowadays, in addition to regular user devices such as laptops, phones, and tablets, many IoT devices such as security cameras, smart locks, and medical devices rely on the 802.11 standard. Several reasons support the importance of this standard for IoT applications: First, compared to 802.15.4 and BLE, the 802.11 standard offers higher data rates, and compared to cellular communication, 802.11 operates in unlicensed bands. In addition to reducing costs, these features are particularly useful in domains such as video surveillance, industrial control, and medical monitoring, where high bandwidth is necessary. Second, 802.11 base stations—known as Access Points (APs)—are broadly deployed in various environments and provide an infrastructure for IoT connectivity. Third, the power consumption of 802.11 transceivers has been considerably reduced during the past decade by introducing various power-save mechanisms and developing low-power RF transceiver technologies such as power gating and clock gating [1]–[3]. Nowadays, 802.11-based IoT devices are prevalent in the market, such as *Ring* security camera and doorbell, *Nest* security camera, *Schlage* locks, and *LIFX* light bulbs, to mention a few. Nevertheless, achieving both energy efficiency and timeliness using the 802.11 standards is more challenging compared to 802.15.4 and BLE. Specifically, 802.15.4 and BLE are used in scenarios where the properties of data flows are known (e.g., WirelessHART), or when the data exchange size is small and sporadic (e.g., Nest BLE temperature sensors). In contrast, for example, in a smart home scenario, an increasing number of devices share a wireless infrastructure to dynamically exchange various types of flows such as voice, video, and background.

The 802.11 standard offers multiple power-saving mechanisms to support energy-constrained stations. Power Save Mode (PSM) enables the stations to wake up periodically and check if the AP has any buffered packet(s) for them. The AP periodically broadcasts beacon packets at a certain interval, called Beacon Interval (BI), to inform the stations about their buffered packets. Stations send PS-Poll packet to the AP to request downlink delivery. PSM significantly increases communication delay because stations can only receive downlink packets after each beacon instance. The delay problem further exacerbates with the concurrent transmission of PS-Poll packets and the accumulation of downlink packets after each beacon instance [4], [5]. To reduce communication delay with AP, Adaptive PSM (APSM) requires a station waiting for downlink packets to stay awake for a *tail time* duration (e.g., 10 ms [1]) after each packet exchange [6]. The tail time may cause idle listening and energy waste, especially if the delay between uplink and downlink delivery is longer than tail time. Another enhancement of PSM is Automatic Power Save Delivery (APSD), which is available in 802.11n, ac, and ax. APSD allows stations to request downlink packet delivery by sending NULL packets to the AP [7]. A new power-saving mechanism introduced in 802.11ax is Target Wake-up Time (TWT), which was primarily introduced in 802.11ah for low-power IoT communication. Using TWT, pairwise agreements between AP and stations can be established, and stations are allowed to skip receiving beacon packets. To further reduce the overhead of periodic wake-ups, the new 802.11ba standard specifies the addition of a low-power Wake-up Radio (WUR) [8], [9]. The primary radio only wakes up when the station receives a command over the WUR or when the station needs to perform uplink transmission. Although these power-saving mechanisms allow

stations to reduce their energy consumption significantly, they do not consider the effect of communication delays caused by buffering, interference, channel access method, and traffic category.

Many IoT applications require the transmission of uplink reports by station and reception of commands from a server. For example, consider a sample medical application where an IoT device reports an event and expects to receive actuation commands in return. Another example is a security camera that transfers an image whenever motion is detected and waits for a command to stream video if a particular object is detected. After the transmission of uplink packet(s), the IoT station has five options before receiving downlink packet(s):

– (i) use Continuously Active Mode (CAM),
– (ii) use PSM and return to sleep mode and wake up during the next beacon period,
– (iii) use APSM and stay in awake mode for a short time duration,
– (iv) use APSD or TWT and periodically check if the downlink packet has arrived,
– (v) use APSD or TWT and wake up when the downlink packet is about to be delivered.

Case (i) minimizes delay but does not offer any power efficiency. Case (ii) causes long end-to-end delays [10], [11] because the station has to wait until the next beacon instance, even if the actual downlink delivery delay is less than the time remaining until the next beacon. Besides, the delay considerably increases when the station and server need to complete multiple rounds of packet exchange to make a decision[1]. Case (iii) is effective if the delivery delay is short; otherwise, this case results in power waste in tail time. Case (iv) results in periodic wakeups and unnecessarily increases channel access contention. Therefore, none of these cases are suitable for applications where both delay and energy efficiency are the essential performance metrics. Applying case (v) *requires an accurate estimation of the delay between uplink and downlink packets*. This delay is composed of the following components: First, the uplink packet received over the wireless interface must be sent over the wired interface. The second component is the interval between the instance the packet leaves the AP until a response is received from the server. Third, once the reply is received, the packet must compete with other downlink packets and be delivered to the station in awake mode. In this paper, we show that computing the third delay component is particularly challenging because it depends on various factors, including channel utilization, the intensity of uplink and downlink communication, access category of packets, and AP's buffer status. This is also verified by the recent studies that show the delay experienced at AP is more than 60% of the total communication delay between a station and server [13], [14]. Besides, the buffering mechanisms employed in Linux's queuing discipline (qdisc) and wireless driver further complicate the modeling and prediction of these delay components [14]–[17]. For example, Intel's IWL and Qualcomm's ath9k and ath10k drivers perform packet scheduling; however, these algorithms are heuristically designed by vendors—further complicating delay estimations.

1. Considering user interactions, studies show that every 10 ms increase in network access results in a 1000 ms increase in page load time [12].

In this paper, we propose a novel mechanism called *edge-assisted predictive sleep scheduling* (EAPS) to reduce the idle listening time and energy consumption of stations when waiting for downlink packets. At a high level, EAPS works as follows: Once an uplink packet is received from an IoT station, the delivery delay is computed using machine learning techniques. The estimated delay is then conveyed to the station using a high-priority data-plane queue. The station then switches into sleep mode and wakes up at the scheduled time to retrieve downlink packet.

This paper introduces the following contributions: *First*, we present the implementation of a Linux-based AP with new and modified kernel and user-space modules to keep track of the system operation in terms of parameters such as incoming and outgoing transmissions, buffer status, and channel utilization. *Second*, we introduce multiple traffic characterization methods. Using the modified AP, we build a configurable testbed that allows us to generate various traffic patterns similar to real-world deployments. *Third*, the information collected across the 802.11 stack is conveyed into a user-space module to estimate the delay components. We focus on wired-to-wireless switching delays and their prediction using various machine learning algorithms under different traffic scenarios. EAPS runs at network edge to ensure quick sleep schedule computation, which implies that all the necessary computations to calculate sleep schedules are performed by AP to avoid any additional processing overhead on resource-constrained IoT stations. Therefore, regardless of the IoT station's processor type, EAPS can be employed by any 802.11-based IoT devices. *Fourth*, we perform an empirical evaluation of delay prediction and its impact on energy efficiency and timeliness in scenarios where IoT stations communicate with cloud and edge servers. In terms of delay, EAPS outperforms PSM by 45% in the cloud scenario and by 84% in the edge scenario. The energy consumption of EAPS is 26% lower in the cloud scenario and 6% in the edge scenario, compared to PSM. In the edge scenario, the delay of EAPS is close to that of APSM, while its energy efficiency is improved by 37%. In the cloud scenario, EAPS improves delay and energy efficiency by 41% and 46%, respectively, compared to APSM.

The rest of this paper is organized as follows. We overview the related work in Section 2. We present delay components and implementation details of the AP in Section 3. We present the edge-assisted sleep scheduling mechanism in Section 4. Section 5 presents empirical performance evaluations. Section 7 concludes the paper.

2 Related Work

Peck et al. [18] propose PSM with adaptive wake-up (PSM-AW), which includes a metric called *PSM penalty* to enable the stations to establish their desired energy-delay tradeoff. The authors define server delay as the total delay between sending a request to a server and receiving a reply. Based on Round-Trip Time (RTT) variations, the sleep duration is dynamically adjusted to satisfy the desired tradeoff. Also,

the size of the history window of server variations is dynamically tuned based on the range of server delay variations. Compared to our work, PSM-AW [18] only considers AP-server RTT, thereby ignoring the variable and long impact of downlink wireless communication delay. Besides, since RTT sampling and averaging are performed by stations, it adds additional load on resource-constrained stations. Furthermore, delay estimation is directly affected by station-server communication, and estimation accuracy drops as the interval between transactions increases. In contrast, our work does not impose overhead on stations, and once a model is trained, it does not rely on ongoing communication to compute sleep schedules. Jang et al. [6] proposed an adaptive tail time adjustment mechanism by relying on inter-packet arrival delays. A moving average scheme is used to predict inter-packet arrival delay when a burst of packets arrives at a station. The station stays in the awake mode if the next packet arrival time is before the tail time expiry. If packet delivery is after the expiry, the station may extend its tail time based on the outcome of an energy-delay tradeoff model. In contrast to our work, neither [18] nor [6] considers the impact of buffering and channel access delay as variable, essential components of downlink delivery delay. Furthermore, the effectiveness of these approaches highly depends on the burst length and variability of end-to-end delays. Specifically, the moving average scheme employed in [6] would not be effective in IoT scenarios where most of the bursts are short-lived. Sui et al. [19] propose WiFiSeer, a centralized decision-making system to help stations choose the AP providing the shortest delay. WiFiSeer works in two phases: During the learning phase, a set of parameters (such as RSSI, RTT) are pulled every minute from all APs using SNMP. Then a random forest model is trained to generate a two-class learning model for classifying APs into high latency and low latency. A user agent installed on smartphones communicates with the controller and associates the station with the recommended AP. WiFiSeer is vertical to our solution to further reduce station-AP delay.

Jang et al. [20] study the overhead of radio switching and show that stations can achieve significant energy saving during inter-frame delays while the AP is communicating with other stations. The proposed AP-driven approach, called Snooze, utilizes the global knowledge of the AP to schedule sleep and awake duration of each station based on inter-packet delays and traffic load of the station. To distribute the schedule, the AP needs to exchange control messages with stations. Compared to Snooze, our work considers the sensing-actuation pattern of IoT applications and reduces the idle listening time between sensing and actuation. In addition, our work takes into account the impact of interference by measuring channel utilization perceived by the AP. Also, Snooze does not benefit from APSD. Sheth and Dezfouli [21] propose *Wiotap*, an AP-based packet scheduling mechanism for IoT stations that employ APSM. This mechanism uses an Earliest Deadline First (EDF) scheduling strategy to maximize the chance of packet delivery before tail time expiry. Rozner et al. [10] propose NAPman, which prioritizes the delivery of PSM traffic as long as other stations are not affected. Tozlu et al. [3] show that increasing AP load has a higher impact on packet loss and RTT, compared to out-of-band interference. Pei et al. [13] demonstrate that station-AP link delay comprises more than 60% of station-server RTT. They also show that more than 50% of packets experience a delay longer than 20 ms over station-AP links. This delay is longer than 100 ms for 10% of packets. For TCP traffic, the authors proposed an approach to measure the delay of wired latency as well as uplink and downlink channel access delay. Using the Kendall correlation, they also show that channel utilization, RSSI, and retry rate are the top three factors affecting station-AP delay. They used a decision tree model to tune the parameters of APs and reduce the overall delay experienced by stations. This is in contrast to our work, which offers per-station and fine-grained sleep scheduling. Primarily designed for VoIP traffic, Liu et al. [22] propose a mechanism to reduce contention among stations waking up using APSD to retrieve packets from the AP. After receiving a burst of voice data, the station measures the tolerable deadline of incoming packets and informs the AP about its wake up time before switching to sleep mode. The wake-up instance is approved only if the AP will be idle when the station wakes up.

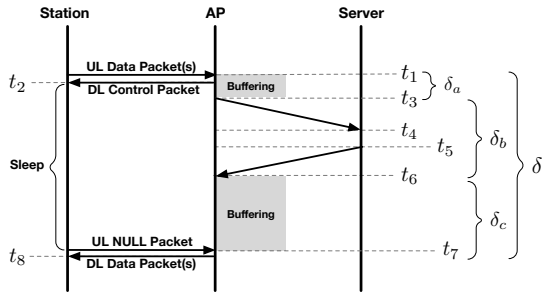

Fig. 1. The end-to-end delay components between a station and a server. The prediction of δ_c is particularly challenging because it is affected by several factors such as traffic rate, channel utilization, and buffering mechanisms employed by Linux's qdisc and wireless NIC's driver.

3 Delay Analysis and AP Development

3.1 Delay Components

As Figure 1 shows, at time t_1 the station grasps the channel and transmits its uplink packet. This uplink packet may represent a single uplink packet sent by the station or the last packet of a burst of uplink packets. After this, the station waits to receive downlink packet(s) from the AP. We refer to the process of uplink and downlink packet exchange as a *transaction*. In event-driven applications, the downlink packet is usually a command message issued by a server in response to the message sent by the station. As discussed in §1, multiple power saving strategies can be used by the station to save power while waiting for the downlink packet. The goal of this paper is to inform the station about the delivery time of downlink packet. Therefore, we enable the station to switch to sleep mode and wake up when the downlink delivery is about to happen.

To reduce the waiting time for downlink packet delivery, the station transitions into sleep mode after the reception of a control packet at t_2 and wakes up at t_7 to request and

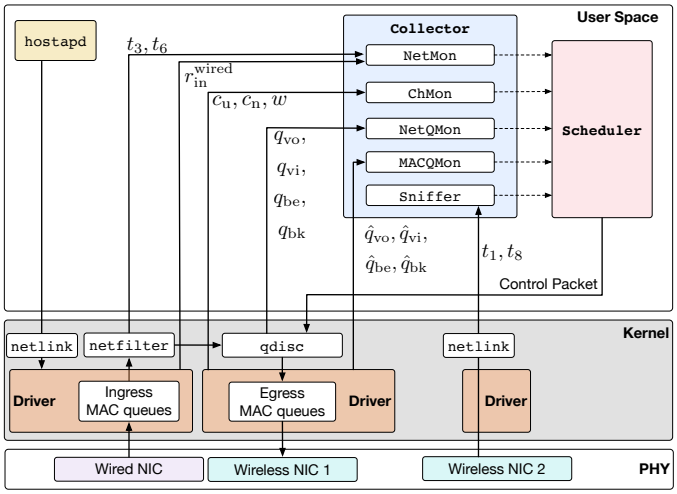

Fig. 2. The AP architecture developed and used in this paper. The `Collector` module communicates with various kernel and user-space components to collect a set of features required for delay prediction. The `Scheduler` estimates the sleep duration and conveys it to the station. This figure primarily focuses on the wired-to-wireless interfaces path to compute δ_c. Some of the modules required to collect other delay components (δ_a and δ_b) are not included in this figure.

receive the downlink packet. The sleep duration is conveyed to the station by the AP through a control packet sent at t_2. Therefore, we need to estimate the delay between t_1 to t_7. To this end, we first modify a Linux-based AP.

3.2 AP Development

The current AP architectures do not provide the necessary tools to collect and apply predictive scheduling [23]. In this section, we present an AP architecture that allows us to collect the features necessary for predictive scheduling. Figure 2 presents the modules we developed on a Linux-based AP. The user-space components of the AP are `Collector` and `Scheduler`. The `Collector` is responsible for collecting all the features required to predict delay. This information is then shared with and used by the `Scheduler` to train a model, estimate the sleep duration, and dispatch the schedule. The information collected by the `Collector` is stored in the physical memory (using `mmap`) to reduce data access delay. The `Collector` module includes the following modules: The `Sniffer` module utilizes the `libpcap` library to capture the timestamp of packets as soon as they are sent or received by the wireless NIC. The `NetMon` module records packet exchange instances over the wired interface as well as incoming data rate over this interface. The `NetQMon` and `MACQMon` are responsible for keeping track of the utilization of qdisc and MAC layer queues, respectively. The `ChUMon` module captures channel utilization.

To perform the standard AP functionalities, we use `hostapd` [24], which is a user-space daemon that handles beacon transmission, authentication, and association of stations. The underlying hardware includes an Atheros AR9462 wireless NIC, ath9k driver, a Core i3 processor, and 8 GB of RAM. The AP operates in 802.11n mode, uses two antennas, and supports up to 144 Mbps. We explain the implementation detail of the AP in the next three sections.

3.3 Communication Delay Between AP and Server

Once the AP receives an uplink packet, it is stored in the qdisc of wired interface, then the packet is sent over the wired interface. The qdisc is the scheduling mechanism employed by the kernel to schedule the transmission of packets while switching them between two interfaces. This buffering delay, denoted as $\delta_a = t_3 - t_1$, depends on the difference between the rate of incoming wireless uplink packets (destined to the wired interface) and the rate of transmitting these packets over the wired interface. The primary types of networks considered in this paper are smart home environments where an AP is connected to an Internet modem, and campus and business deployments where APs communicate via an Ethernet infrastructure. In such networks, the speed of APss' wired interface is fixed and usually higher than the wireless interface. This are reasonable assumptions because: First, in enterprise environments, APs are connected to switches via Ethernet links supporting at least 1 Gbps. This may also be true in a residential environment where the AP is connected to a local processing server through Ethernet [25]. For residential environments, also, cable modems and fiber-to-the-home (FTTH) provide data rates higher than wireless. Second, the uplink speed between a home modem and an Internet provider is fixed. For example, DOCSIS employs a combination of TDMA and CDMA for deterministic channel access.

Based on these observations, we estimate δ_a, denoted as δ'_a, by using the number of packets currently in the qdisc of wired interface (*not shown* in Figure 2). We have modified the qdisc module to communicate the number of packets in this buffer with `NetMon`. For each packet p_i in qdisc, the `Scheduler` computes $\psi(p_i) = 8 \times (s(p_i) + h_{\text{mac}} + h_{\text{phy}})/l_{\text{out}}^{\text{wired}}$, where $\psi(p_i)$ is the time required to transmit p_i, $s(p_i)$ is the packet size (bytes), h_{mac} is the MAC header size (bytes), h_{phy} is the physical header size (bytes), and $l_{\text{out}}^{\text{wired}}$ is the transmission bit rate supported by the wired link. The switching delay is therefore computed as $\delta'_a = \sum_{\forall p_i \in \text{qdisc}} \psi(p_i)$.

The delivery delay between the AP and the server, i.e., $t_4 - t_3$ and $t_6 - t_5$, depend on various factors and primarily on the number of hops between these two nodes. Based on this number, we consider *edge* and *cloud* computing scenarios. Edge computing is employed in latency-sensitive and mission-critical applications to minimize the latency and overhead of communication over the wired network [26]. In the cloud computing scenario, the server is located at least a few hops away from the AP. For both cases, to measure this delay (denoted as δ'_b), we use a moving average, which is the standard approach used by various protocols such as TCP to estimate RTT [27], [28]. To this end, we modify the `netfilter` [29] kernel module to communicate with the `NetMon` module and timestamp t_3 as the instance the packet is sent to the wired NIC, and t_6 as the instance the packet has arrived in the AP.

3.4 AP to Station Delivery Delay

An incoming packet from the wired interface first passes through ingress driver queues. Subsequently, the packet is processed by the `netfilter` module. The packet is then

queued in the qdisc. Finally, the packets are queued in the Enhanced Distributed Channel Access (EDCA) queues inside the wireless NIC's driver. These packets are served according to the channel contention parameters specified by the 802.11e standard. Each driver queue contends (individually) for channel access before packet transmission.

Here we mainly focus on the delay between the arrival of a downlink packet through the AP's wired interface and its transmission through the wireless interface. This delay is denoted as $\delta_c = t_7 - t_6$. It is particularly challenging to model and predict this delay because it is affected by several factors such as queuing strategy and queue utilization at the qdisc and MAC layer, channel utilization, number of stations, and link quality. However, in addition to the high complexity of buffering mechanisms implemented by wireless drivers such as ath9k and ath10k [15], [16], the actual operation of non-open source drivers is not known, which makes it impossible to develop a mathematical model of buffering delay. Therefore, we follow a data-driven approach to predict δ_c. The predicted value is denoted as δ_c'. The `Collector` module time stamps the switching delay between the wired and wireless interfaces. The time of packet arrival from the downlink transmission is determined by the `Sniffer` module, which in turn informs the `Collector`. During t_6 to t_8, the `Collector` also collects statistics regarding the status of queues and channel condition. The collected parameters are explained in the following sections.

3.4.1 Input traffic rate through wired interface

The incoming traffic through wired interface, denoted as $r_{\text{in}}^{\text{wired}}$ (bytes/second), impacts the current and future utilization of wireless interface's qdisc and driver queues. Hence, the `NetMon` module communicates with wired interface's driver to collect incoming traffic rate.

3.4.2 qdisc queues

Every network interface is assigned a qdisc, which is pfifo_fast by default [14]. This mechanism contains three bands, and dequeuing from a band only happens when its upper bands are empty. The PRIO qdisc is a classful configurable alternative of pfifo_fast and enables us to configure the number of bands. To enqueue the packets of each Access Category (AC) in its own queue, we implement four queues in this layer. These queues are denoted by $\mathbf{Q} = \{q_{\text{vo}}, q_{\text{vi}}, q_{\text{be}}, q_{\text{bk}}\}$. We have modified the PRIO kernel module to communicate with the `NetQMon` module to collect the number of packets in each qdisc band.

With PRIO qdisc, the queuing delay experienced by a packet enqueued in the lowest priority queue not only depends on the current utilization level of that queue, but also on the number of packets in the higher priority queues. In addition to the four queues mentioned above, we have also included an additional queue—called *control queue*—that is assigned the highest priority level. We will utilize this queue in order to implement the highest-priority data plane used to send the *control packet* that conveys sleep schedules to stations. We will explain this packet later. It is worth mentioning that, although our implementation utilizes the PRIO qdisc (the default policy used in several Linux distributions), the concept can easily be extended to other types of qdiscs, such as Hierarchical Token Bucket (HTB).

3.4.3 Wireless channel condition

Both interference and channel utilization are the main channel condition parameters that affect packet transmission delay. The duration and intensity of these parameters depend on various factors, such as the number of contending stations and APs, burst size, TXOP, and the transmission power of nearby stations and APs. Therefore, accounting for the effect of channel condition through measuring the factors (mentioned above) would be very challenging. Instead, we collect three parameters to capture the effect of interference and channel utilization on the delay of packet transmission. The first parameter is channel utilization (c_{u}), which refers to the amount of time the AP or its associated stations are transmitting. The second parameter is the number of MAC layer retransmissions (w) performed by the AP to deliver packets to stations. The third parameter is the channel's noise level (c_{n}), which reflects the activity of nearby wireless devices (such as other APs and stations, ZigBee, and Bluetooth devices).

Most 802.11 drivers maintain counters that represent channel utilization rate. For example, the rate of updating `ch_time_busy` reflects channel utilization during a sampling interval. The `ChUMon` module is responsible to extract these counters from the driver. We realized that the interval of obtaining channel utilization impacts measurement accuracy. We obtained the peak accuracy, in terms of Kendall's correlation coefficient, when the frequency of polling c_{u} is 10 ms. Additionally, the granularity of the measurements also decreases as we increase the frequency of polling channel utilization. This is because the counters use millisecond granularity. For example, if the sampling frequency is 10 ms, the granularity of c_{u} obtained in percentage is 10%.

3.4.4 Driver's transmission queues

Using Enhanced Distributed Coordination Function (EDCF), packets arriving at the MAC layer are categorized and inserted into one of the four queues assigned to each station inside the driver. The categorization relies on the IP header's Type of Service (ToS) field. These queues are denoted by $\hat{\mathbf{Q}} = \{\hat{q}_{\text{vo}}, \hat{q}_{\text{vi}}, \hat{q}_{\text{be}}, \hat{q}_{\text{bk}}\}$. Each queue behaves like a virtual station that contends for channel access independently. In case of internal collision between two or more queues, the higher priority queue is granted the transmission opportunity. The status of these queues are monitored by the `MACQMon` module through communicating with the driver. Considering the size of these queues allows the prediction models to account for the effect of packet aggregation and packet bursting. Specifically, for each AC, packet aggregation is applied if a station's queue includes more than one packet. Also, depending on the AC, a burst of packets may be sent without contending for the channel on a per-packet basis.

3.4.5 Summary of the features collected

The `Scheduler` interacts with `Collector` to gather the features necessary for delay prediction. In summary, the developed AP enables us to collect the following features periodically:

$$\mathbf{C_u}, \mathbf{C_n}, \mathbf{R}_{\text{in}}^{\text{wired}}, \mathbf{W}, \mathbf{Q}_{\text{vo}}, \mathbf{Q}_{\text{vi}}, \mathbf{Q}_{\text{be}}, \mathbf{Q}_{\text{bk}}, \widehat{\mathbf{Q}}_{\text{vo}}, \widehat{\mathbf{Q}}_{\text{vi}}, \widehat{\mathbf{Q}}_{\text{be}}, \widehat{\mathbf{Q}}_{\text{bk}} \quad (1)$$

where $\mathbf{C}_{\text{u}}$, $\mathbf{C}_n$, $\mathbf{R}$, $\mathbf{W}$, $\mathbf{Q}$, and $\widehat{\mathbf{Q}}$, represent the list of channel utilization values, list of channel noise values, list of incoming traffic rate values over wired interface, list of MAC layer downlink retransmission values, list of the utilization values of qdisc queues (for each AC), and list of the utilization values of driver queues (for each AC), respectively. Each list includes periodically collected values. For example, assuming that each list contains $k+1$ values, list $\mathbf{C}_u$ is represented as follows:

$$\mathbf{C}_{\text{u}} = [c_{\text{u}}(t' - k \times \Delta), c_{\text{u}}(t' - (k-1) \times \Delta), ..., c_{\text{u}}(t' - \Delta), c_{\text{u}}(t')] \quad (2)$$

where $c_{\text{u}}(t')$ is the last sampled channel utilization value, and Δ refers to sampling interval. In our implementation, $\Delta = 10$ ms. We did not use a shorter sampling interval because of the significant increase in processor utilization ($> 30\%$). Implementing a more efficient AP architecture is a future work.

In addition to the features collected periodically, we add two features that are collected once per prediction. First, since each AC uses its own channel access and Transmit Opportunity (TXOP) parameters, we include the AC of the transaction as a feature. Second, we use $\delta'_a + \delta'_b$ because the predicted delay (δ'_c) depends on the interval between the uplink packet and the arrival of downlink packet over the wired interface. For example, if the server delay is expected to be 30 ms, the prediction for δ_c should be made for a packet that would arrive at the AP in 30 ms.

3.5 Schedule Announcement

When a predicted delay value is computed, the `Scheduler` creates a UDP control packet to send the value to the station. This packet includes δ'_a, δ'_b, and δ'_c, as well as the standard deviation of prediction error, where each value is encoded as one byte. The value of each byte reflects duration in milliseconds. This data packet is sent using a dedicated queue with highest priority. When this control packet reaches the station at t_2, the station immediately transitions into sleep state for $\delta'_a + \delta'_b + \delta'_c - (t_2 - t_1)$. Note that the AP shares the *relative* wake-up time with the station. Since the AP has computed wake-up schedule at time t_1, the station needs to measure $t_2 - t_1$ and subtract it from the shared value. The station can simply use a timer to measure $t_2 - t_1$.

At the end of the sleep interval, the station wakes up and informs the AP about its transition into awake mode. This is achieved by relying on APSD, which is supported by the state-of-the-art wireless NICs. To this end, at t_7 (in Figure 1), the station wakes up and sends a NULL packet to the AP, conveying that the station is ready for receiving a packet. The AP responds by sending one or multiple downlink packets starting at t_8. As per the 802.11e amendment, multiple packets can be sent during a Transmission Opportunity (TXOP) without having to contend for channel access. For example, if the traffic belongs to the voice AC, the AP uses a 1.504 ms slot (as long as packets exists) for downlink delivery.

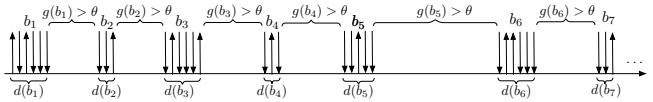

Fig. 3. Traffic characterisation.

4 Predictive Scheduling

In this section, we first present traffic characterization methods and our testbed setup, which are then used for realistic traffic generation, model training, and evaluation. We also discuss the various stages of the statistical learning and modeling process and empirically study the performance of multiple machine learning algorithms in terms of delay prediction accuracy.

4.1 Traffic Generation

As explained in the previous sections, AP modification is necessary to collect the features required for predictive scheduling. Also, we need to introduce controlled changes in the traffic pattern of the environment to study the impact of these changes on prediction accuracy. Therefore, it is required to have a testbed that represents the traffic pattern of real-world environments as well as controllability over traffic generation parameters. To achieve this, we systematically characterize and compare the scenarios generated in our testbed with those collected in real-world environments.

A *burst*, denoted as b_i, is defined as a train of packets in either UL or DL direction with inter-arrival time less than a threshold value θ [30]. Resembling 802.11 traffic, Figure 3 illustrates a series of bursts. The duration (in seconds) of a burst b_i is denoted by $d(b_i)$. $g(b_i)$ refers to the gap (in seconds) between two consecutive bursts b_i and b_{i+1}.

To generate traffic flows representative of various levels of *network dynamics* in real-world environments, we have developed a testbed that includes two types of stations: (i) stations such as laptops, smartphones, and IoT devices, and (ii) four Raspberry Pi boards to control traffic generation pattern. Each RPi runs four threads, where each thread can be involved in a downlink, uplink, or bidirectional flow. This enables us to introduce up to 16 additional controlled flows into the network. The implementation of traffic control capability is composed of a set of Python scripts that use the iperf tool under the hood. A central controller is in charge of setting the parameters of traffic flows. Among the flow parameters, we can modify AC, transport layer protocol, packet size, bit rate, burst size, and inter-burst interval. Also, we note that sharing flow characteristics by consecutive flows is more likely. To represent this behavior, after each burst, the controller either repeats the process of traffic selection or chooses the same parameters for the next burst based on a *variability parameter* denoted by ν. Specifically, a higher value of ν results in a higher dynamicity. Hence, we use $\nu = 0.9$ to generate *high dynamicity* (HD) traffic, and $\nu = 0.1$ to generate less diverse traffic referred to as *normal dynamicity* (ND). Also, for voice and video ACs, UDP is preferred because it is the dominant transport protocol for real-time traffic.

As demonstrated in [30], capturing network dynamics can be achieved by focusing on characterizing burstiness.

Additionally, Xiao et al. [31] characterized a flow as *regularly* bursty when the standard deviation of the inter-burst intervals ($g(\cdot)$ seconds) and burst sizes ($s(\cdot)$ bytes) are relatively smaller. Otherwise, the flow is characterized as *randomly* bursty. However, based on Xiao's metrics for burstiness, traffic with fewer bursts per unit time can still have a high standard deviation of $s(\cdot)$ and $g(\cdot)$. Hence, we consider burst frequency (i.e., number of bursts per second) and burst size for calculating traffic burstiness. Additionally, due to the difference in the scale of those two parameters, we normalize the burst rate (per second) in the range $[0, 1]$. We define traffic *burstiness*, denoted by $\mathcal{B}$, as follows:

$$\mathcal{B} = \left(1 - \frac{1}{\mathcal{M}}\right) \times \left(\frac{\sum_{i=1}^{N} s(b_i)}{N}\right) \tag{3}$$

where N is the number of bursts in the dataset, $s(b_i)$ is the size of burst b_i (in bytes), and $\mathcal{M}$ is average number of bursts per second.

In addition to traffic burstiness, we define another metric that represents traffic *dynamicity* based on various aspects including burst size, burst duration, inter-burst interval, and the AC of the packets in each burst. This metric, which we refer to as *dynamicity* and is denoted by $\mathcal{D}$, is defined as follows:

$$\mathcal{D} = \frac{1}{N}\sum_{i=2}^{N} \frac{\frac{|d(b_i)-d(b_{i-1})|}{d(b_{i-1})}}{g(b_{i-1})} + \frac{1}{N}\sum_{i=2}^{N} \frac{\frac{|s(b_i)-s(b_{i-1})|}{s(b_{i-1})}}{g(b_{i-1})} + \frac{1}{N}\sum_{i=2}^{N} \frac{\frac{|p(b_i)-p(b_{i-1})|}{p(b_{i-1})}}{g(b_{i-1})} + \frac{1}{N}\sum_{i=2}^{N} \frac{z(b_i)}{g(b_{i-1})} \tag{4}$$

where,

$$z(b_i) = \frac{|p_{\text{vo}}(b_i) - p_{\text{vo}}(b_{i-1})|}{p_{\text{vo}}(b_{i-1})} + \frac{|p_{\text{vi}}(b_i) - p_{\text{vi}}(b_{i-1})|}{p_{\text{vi}}(b_{i-1})} + \frac{|p_{\text{be}}(b_i) - p_{\text{be}}(b_{i-1})|}{p_{\text{be}}(b_{i-1})} + \frac{|p_{\text{bk}}(b_i) - p_{\text{bk}}(b_{i-1})|}{p_{\text{bk}}(b_{i-1})} \tag{5}$$

Here, $p_x(b_i)$ is number of packets belonging to an AC x in a burst b_i. Parameter $z(b_i)$ reflects the change in the number of packets belonging to each AC in each burst compared to that in the previous burst.

4.2 Data Collection

We use the metrics mentioned above and compare datasets generated in our testbed against those collected in multiple real-world environments. Figure 4 presents the results. In general, we observe that the ND scenario resembles real traffic. The HD scenario offers higher network dynamics, which is essential to study the robustness of predictive scheduling.

When generating data in our testbed, the type of each transaction is selected from the voice, video, background, and best-effort ACs with equal probability. The inter-transaction delays are uniformly distributed between 1 ms and 500 ms. In addition to the features discussed in §3, we also collect δ_a, δ_b, and δ_c values per transaction. We split each dataset, such that 70% of it is used for training and the remaining 30% is used for validation. We use independent datasets, referred to as the test datasets, consisting of 10,000 data points for evaluating the performance and robustness of each modelling approach in the ND and HD scenarios.

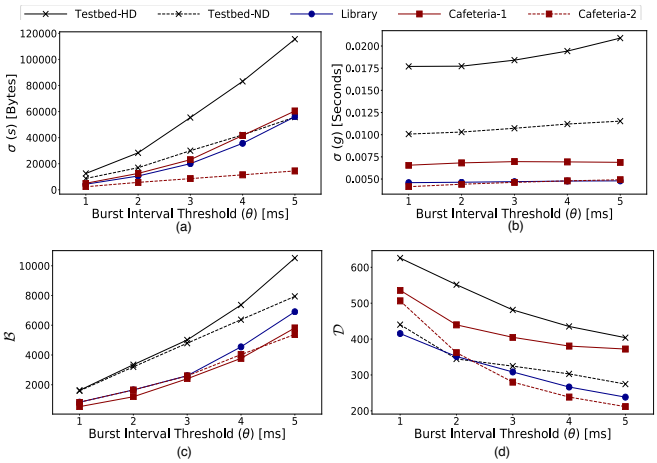

Fig. 4. (a) Standard deviation of burst size, (b) standard deviation of burst interval, (c) burstiness ($\mathcal{B}$), and (d) dynamicity ($\mathcal{D}$) of traffic generated by our testbed compared to traffic captured in real-world environments. ND and HD refer to normal and high dynamicity, respectively.

4.3 Data Pre-processing

We focus on delay prediction for $\delta_c < 100$ ms, for two reasons: First, considering edge computing scenarios, observing RTTs more than 100 ms is very unlikely. Second, almost all commercial APs implement 102.4 ms as their beaconing period. Therefore, all stations wake up every 102.4 ms to synchronize with AP beacons and check if the AP has any buffered packets.

The feature set varies in terms of ranges and units. For example, c_{u} varies from 0 to 100%, whereas c_{n} varies from -95 to -66 dBm. Since this would result in disproportional treatment of the features by the machine learning algorithms, we scale each of the features into the range $[-1, +1]$. Furthermore, the dataset contains more samples (transactions) with delay range $[1, 50]$ ms, compared to samples in range $[50, 100]$ ms. We under-sample the majority bins to prevent the algorithms from generalizing the results for the packets whose actual delay is higher.

4.4 Regression Models

Given the continuous nature of the target variable, we identify predicting δ_c as a regression problem. The methods that we use are Random Forest Regressor (RFR), Gradient Boosting Regressor (GBR), Extra Trees Regressor (ETR) and Histogram-Based Gradient Boosting Regressor (HBR), which are widely-used ensemble learning methods for regression. We also use (deep) neural networks, which are more effective in areas such as predicting time-series data.

Referring back to Eq. 2, whenever a prediction must be made, we use: (i) the closest set of features collected at t', and (ii) a weighted average of the last k measurements collected before t'. For example, we use $c_{\text{u}}(t')$ and $c_{\text{u}}(\bar{t}) = \sum_{i=1}^{k} w_i \times c_{\text{u}}(t' - i \times \Delta)$ for channel utilization. Here, $c_{\text{u}}(\bar{t})$ is called the *feature history* of channel utilization, k denotes the length of feature history, and w_i refers to the individual weights assigned to the past feature values such

that $\sum_{i=1}^{k} w_i = 1$. More recent feature values are assigned larger weights. For example, when $k = 2$, $w_1 = 0.75$ and $w_2 = 0.25$. When $k > 2$, $w_1 = 0.5$ and $w_2 = 0.25$ (i.e., half of the remaining weight-budget of 0.5), and this process continues recursively until all k weights are assigned. With this method applied to all the features summarized in §3.4.5, we can capture network dynamics and dependency of the predictions on previous feature history with models that do not support back propagation.

In ensemble learning, final prediction can either be calculated by the average of the predictions of the model trained on random subsets of data (bagging) or calculated via sequentially training the model using prediction success on the previous sample of the dataset (boosting). RFR is an example of the bagging approach and operates by constructing several decision trees during training and makes predictions based on the outputs of the individual trees. RFR runs efficiently on large and high-dimensional datasets. GBR is an example of the boosting approach. Each tree outputs a prediction value at different splits that can be added together, allowing subsequent models to modify error in predictions. HBR is a variant of GBR. Since it is a histogram-based estimator, HBR can reduce the number of splitting points by binning input samples, and therefore improves performance when dealing with large datasets. ETR creates decision stumps at variable tree depths. The features and splits are selected randomly, and are less computationally expensive than other tree-based algorithms.

Neural Networks (NN) have been studied extensively in the past decade for their efficiency in learning complex data features for making predictions. Multilayer perceptrons (MLP) is one such variant of feed-forward neural networks that does not allow feedback loops, thereby resulting in data progressing in a single direction over the network from input to output. One of the biggest drawbacks of using such a network is its lack of memory, i.e., it treats each instance of the input time-series independently and predictions are independent of the history of past inputs to the network. Recurrent Neural Networks (RNN) are a class of neural networks in which the predictions are based on the current and past inputs, and therefore they are suitable for making predictions about δ_c using historical network features. A specific variant of RNN is Long Short-Term Memory (LSTM) [32], which is able to track dependencies of output predictions on input history. The network retains a memory equal to the number of lookbacks that allow the flow of information from the previous timesteps [33]. Lookback is defined as the number of timesteps—*transactions in our particular application*—that are unfolded for back-propagation. Simply put, *transaction history* is the number of previous transactions in the temporal domain that aids in predicting the delay of the current transaction by providing contextual information.

For the ensemble learning methods, we use scikit-learn library and tune the hyper parameters using grid search and a validation dataset to obtain the highest performance on the training data to avoid over-fitting. For the MLP and LSTM model, we use Tensorflow and Keras library [34]. Also, we utilize early stopping mechanism (on the validation dataset) to prevent over-fitting. The optimal MLP model contains five dense layers, each consisting of 32, 20, 16, 10, 8 neurons, respectively, and *ReLu* activation function. The optimal LSTM model contains one LSTM layer followed by three dense layers, each consisting of 20 neurons and *ReLu* activation function. We use a stateless LSTM model, which is the default setting in Keras library. Hence, the inputs to LSTM layer are: (i) hidden cell states that carry information about previous timesteps (transactions), and (ii) feature values of the current timestep. Note that the latter input includes feature values collected at t' as well as the weighted average of the past k measurements. The input to the dense layer (after LSTM layer) is the last hidden state of LSTM layer. While training both the LSTM and MLP models, we tested batch sizes from 10 to 1000. We observed that training duration decreases as the batch size increases. However, we use the batch size of 100 transactions for evaluating the models because the models started overfitting with larger batch sizes. Both the MLP and LSTM models contain an output layer and were trained using Adam optimizer [35] with learning rate of 0.01.

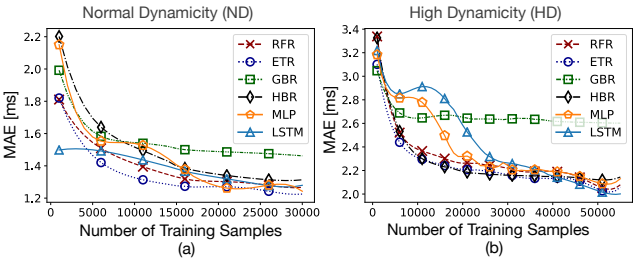

Fig. 5. MAE of machine learning algorithms versus the number of samples (transactions) in training dataset for (a) Normal Dynamicity (ND), and (b) High Dynamicity (HD) scenarios. Results are averaged over all ACs. ETR converges the fastest, and LSTM requires up to 3x more data points compared to ETR.

4.5 Model Evaluation

We used the test dataset for all evaluations. Figure 5 illustrates the Mean Absolute Error (MAE) of δ_c' as a function of the size of training data. For better visibility, we used Savitzky–Golay filter and also added markers at regular intervals in Figure 5. We observed that the performance of ETR converges at the fastest rate, utilizing 15000 and 20000 data points for training under the ND and HD traffic, respectively. Whereas, due to the higher complexity of neural networks, MLP requires 20000 data points in the ND scenario and 35000 data points in the HD scenarios. LSTM requires 30000 data points in the ND and 50000 data points in the HD scenario, thereby showing slower convergence compared to MLP. Based on these results, for the rest of the evaluations presented in this paper, we use the required number of data points that are needed by each algorithm for performance convergence.

Figure 6 quantifies the effect of feature history (k in Eq. 2). For all the algorithms, MAE decreases significantly in both HD and ND scenarios when we include feature history. This decrease continues up to 30 ms, beyond which it does not result in performance enhancement. Feature history helps the model to anticipate features' trend and accurately predict δ_c that would be incurred by the downlink packet shortly. Therefore, in addition to the most recent record,

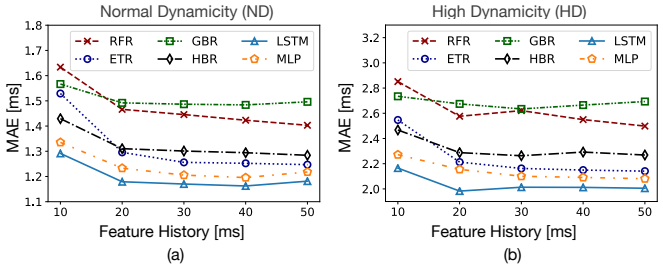

Fig. 6. MAE of machine learning algorithms with respect to feature history in (a) Normal Dynamicity (ND), and (b) High Dynamicity (HD) scenarios. Results are averaged over all ACs.

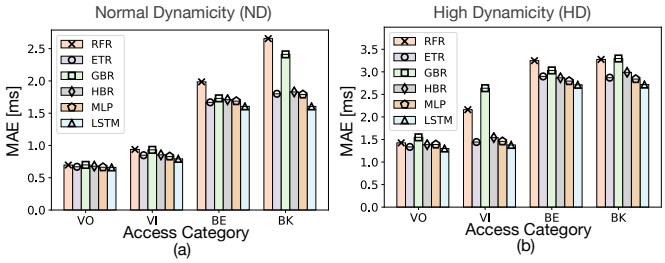

Fig. 7. MAE of machine learning algorithms versus transaction's AC in (a) Normal Dynamicity (ND), and (b) High Dynamicity (HD) scenarios. MAE of VO and VI packets is lower than BK and BE packets.

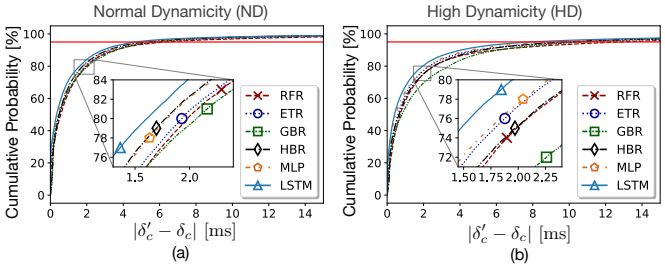

Fig. 8. ECDF of prediction errors ($|\delta_c' - \delta_c|$) while utilizing various machine learning algorithms in (a) Normal Dynamicity (ND), and (b) High Dynamicity (HD) scenarios. All machine learning algorithms are able to predict δ_c' for 95% of the packets with an error of $\pm$5.3 ms in case of ND scenario, and $\pm$10.6 ms for the HD scenario. We have used markers in the inset graph for better visibility.

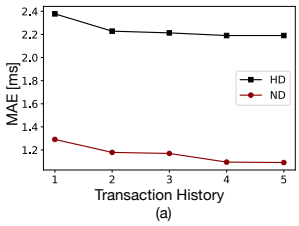

Fig. 9. Effect of transaction history on MAE of LSTM.

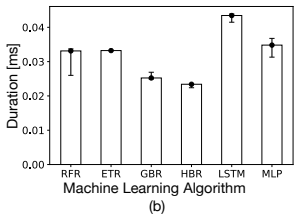

Fig. 10. Processing time of the prediction algorithms.

we include the weighted average of three preceding feature values (corresponding to a total of the preceding 40 ms of feature values) to train the models.

Figure 7 compares the performance of the machine learning algorithms versus the AC of transactions. Averaged over all ACs, the MAE (in millisecond) of algorithms in the ND scenario are: RFR: 1.43, ETR: 1.26, GBR: 1.49, HBR: 1.28, MLP: 1.24, and LSTM: 1.16. For the HD scenarios the MAE values are: RFR: 2.49, ETR: 2.17, GBR: 2.69, HBR: 2.27, MLP: 2.12, and LSTM: 2.01. On average for the ND and HD scenarios, the MAE of LSTM is 14% lower for all ACs, compared to the average MAE of all the other machine learning algorithms.

Figure 7 also shows that the MAE of VO and VI packets is lower than BK and BE packets. The reason is that the packets of these ACs are prioritized over higher ACs at the qdisc layer (using PRIO qdisc) as well as the driver's queues (using EDCA). This results in lower delays incurred by the downlink packets and lower unpredictability caused by the transmission of packets in higher priority queues.

Figure 8 presents the Empirical Cumulative Distribution Function (ECDF) of the deviation of δ_c' from δ_c. The 95th percentile of error ($\delta_c' - \delta_c$) for all machine learning algorithms is less than $\pm$5.3 ms in case of ND scenario and $\pm$10.6 ms for the HD scenario.

Transactions may occur at random time instances and result in irregular time-series. With feature history, we provide the models with a limited amount of historical measurements. For example, if the inter-transaction interval is longer than 40 ms (i.e., feature history of the current transaction), the information about the previous state of the network is not considered in prediction. In this case, using transaction history is particularly beneficial when multiple transactions occur during similar network conditions. Since LSTM predicts based on the current and past transactions' inputs, we estimate the effect of transaction history on the MAE of this model. Figure 9 shows the results. We observe that MAE decreases for up to five lookbacks. This means, on average, five transactions occur during similar network conditions.

Figure 10 presents the prediction execution time of all the machine learning algorithms on a dual-core 2.4 GHz Core-i3 processor. Each marker shows the median of time taken to predict each data point in the test dataset, and the error bars present lower and upper quartiles. We observed that HBR is the fastest (24 μs median and 0.046 μs standard deviation) for prediction, whereas LSTM is the longest (48 μs median and 3 μs standard deviation). However, the time taken to predict the delay in case of LSTM is still considerably shorter than a packet transmission time. For example, with a 1400 bytes packet sent over a 54 Mbps link, the ratio of prediction duration to transmission duration is $48\mu s/207\mu s$.

5 Empirical Evaluation

In this section, we present an empirical evaluation of EAPS versus the power saving mechanisms of 802.11. Since the empirical measurements of prediction accuracy (§4.5) confirm the superiority of LSTM compared to other algorithms, we use this algorithm to compare the performance of EAPS against the power saving methods of 802.11 standard. Note that LSTM requires about 3x more training data for its performance to converge, compared to other algorithms (cf. Figure 5). Hence, in scenarios where it is not possible to collect large datasets for training, either ETR or MLP can be used.

5.1 Testbed

Our testbed includes four IoT stations (cameras and Amazon Echo), four Raspberry Pi boards, regular stations (smartphones and laptops), an AP, and a server. We refer to the IoT stations as *station*. Each station is a Cypress CYW43907 [1], which is a low-power 802.11n SoC designed for IoT applications. To represent a real-world scenario affected by variable background interference, the testbed is located in a residential environment surrounded by APs belonging to multiple households. Also, the four Raspberry Pi boards are used to control network dynamicity and variability in δ_c.

To represent the request-response behavior of IoT traffic, for each uplink packet sent, the server responds by sending a downlink packet back to the station.[2] The exchange of an uplink packet and receiving its response is referred to as a *transaction*. In all the figures of this section, each marker shows the median of 1000 transactions and the error bars present lower and upper quartiles. We use the EMPIOT tool [36] to measure the energy and delay of each transaction. This tool samples voltage and current at approximately 500,000 samples per second. These samples are then averaged and streamed at 1 Ksps. The current and voltage resolution of this platform are 100 μA and 4 mV, respectively.

We use two scenarios to evaluate the performance of EAPS with respect to varying AP-server delays (i.e., δ_b in Figure 1): *edge*, and *cloud* computing. In the former, the server is directly connected to the AP, and in the latter, we use an Amazon AWS server in Oregon, USA. Note that in both cases the sleep schedules are computed at the edge and by the AP the station is associated with.

5.2 Baselines and EAPS Variations

The baselines are PSM, APSM, and CAM. Using PSM, after an uplink packet, the station goes back into sleep mode and wakes up at each beacon instance to check for downlink packet delivery. With APSM, instead of going back into sleep right after packet exchange, the station stays in the awake mode for 10 ms. With CAM, the station always stays in awake mode. *Note that for CAM, we measure only the delay and energy consumption of transactions (only the time interval between the uplink and downlink packets).*

To study energy-delay tradeoffs, we use three versions of EAPS, derived based on observations concerning prediction error. To justify the importance of these three versions, we first present the distribution of prediction errors in Figure 11 for voice and background ACs. Based on the distribution for each AC, the station can either choose to wake up at (i) $\delta' - 2\sigma$, (ii) $\delta' + 2\sigma$, or (iii) δ', where $\delta' = \delta'_a + \delta'_b + \delta'_c$. We call these cases EAPS with Early wake-up (EAPS-E), EAPS with Late wake-up (EAPS-L), and EAPS with Mid wake-up (EAPS-M), respectively. Intuitively, EAPS-E reduces delay with a higher energy consumption, EAPS-L reduces energy with a longer delay, and EAPS-M establishes a tradeoff between energy and delay. Note that EAPS-E is only applicable if $\delta' - 2\sigma > 0$.

2. Note that the case where multiple uplink and downlink packets are exchanged is simply supported as explained in §3.

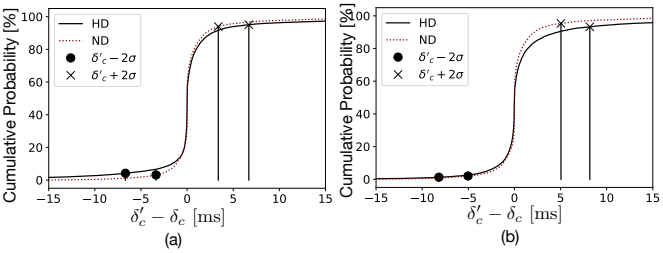

Fig. 11. Cumulative distribution function of prediction error ($\delta'_c - \delta_c$) for (a) voice, and (b) background ACs. Prediction error of voice AC is lower than that of background AC. Depending on the application's energy-delay tradeoff, the station may wake up before, on, or after the predicted time.

5.3 Results

Figures 12 and 13 illustrate the average energy consumption and duration of transactions when the station is communicating with edge and cloud computing platforms under ND and HD conditions, respectively.

In the cloud computing scenario, CAM and EAPS incur an average round trip delay of 35 ms and 42 ms, respectively, while EAPS consumes 63% less energy. This is because EAPS conserves energy expenditure by switching to sleep mode and waking up right before the packet is ready for transmission at AP. In contrast, CAM needs to stay in awake mode until the response is received. Reduction in energy consumption of EAPS compared to CAM reduces to 30% in edge environment due to the shorter duration spent in awake mode to receive the downlink packet.

With PSM, the station immediately transitions to sleep mode after transmitting each uplink packet. While this results in less energy consumption compared to CAM, transactions suffer about 55 ms higher delay on average because the earliest opportunity for downlink packet delivery is after the next beacon instance. The transaction duration of EAPS is 62% lower compared to PSM on average across all the ACs. With APSM, the station remains in idle state for 10 ms after each packet exchange. This is beneficial only in specific scenarios. For example, in the edge scenario, the station receives its downlink packet within the tail time (similar to CAM). However, when the round trip delay is more than 10 ms, the station has to wake up again to retrieve the downlink packet after the next beacon announcement, thereby resulting in higher energy consumption compared to PSM. On average, for both edge and cloud scenarios, the energy consumption of APSM is 30% higher compared to PSM. In contrast, the energy consumption of EAPS is 20% and 43% lower than PSM and APSM, respectively. Also, the transaction duration of PSM, APSM, and EAPS are 77 ms, 10 ms, and 12 ms in edge computing scenario, and 77 ms, 72 ms, and 42 ms in cloud computing scenario.

EAPS allows each node to choose between EAPS-E, EAPS-M, or EAPS-L, according to application requirements. As our results show, with EAPS-E, the station suffers from slightly higher energy consumption because it wakes up early, waits for the packet to be received from the AP, and then transitions into sleep mode. In the case of EAPS-L, since the station wakes up $2 \times \sigma$ after the predicted delay, the probability of immediate packet delivery is higher

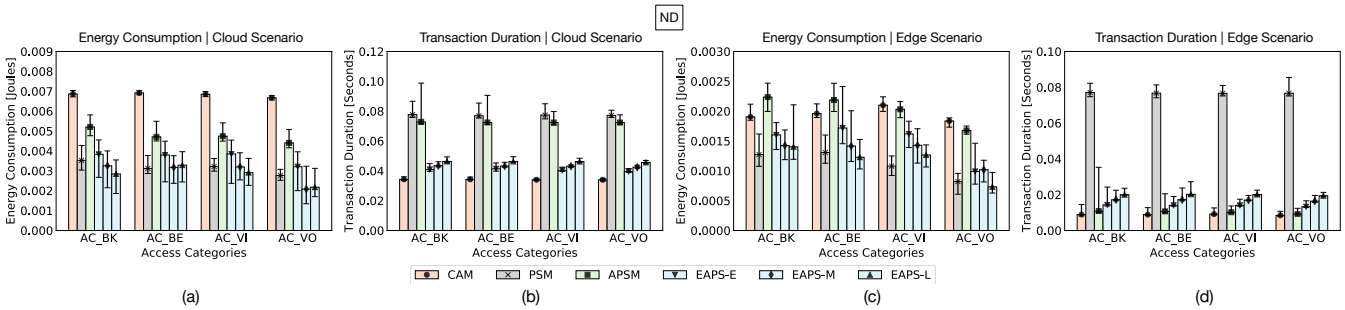

Fig. 12. Performance comparison of EAPS with 802.11 power saving mechanisms in *ND* conditions for all ACs. (a) and (b) show the average *per-transaction* energy and duration in cloud scenario, respectively. (c) and (d) show the average *per-transaction* energy and duration in edge scenario, respectively.

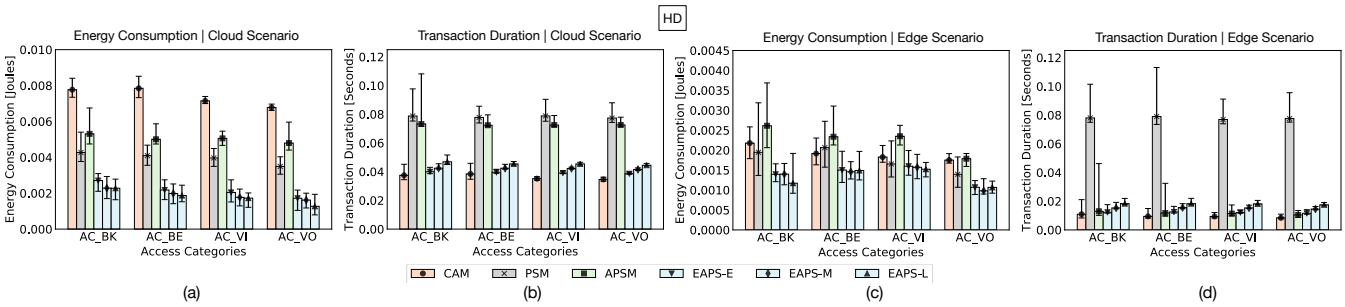

Fig. 13. Performance comparison of EAPS with 802.11 power saving mechanisms in *HD* conditions for all ACs. (a) and (b) show the average *per-transaction* energy and duration in cloud scenario, respectively. (c) and (d) show the average *per-transaction* energy and duration in edge scenario, respectively.

once the station wakes up, and the station can immediately transition to sleep mode once the packet is received. Thus, energy consumption of EAPS-L is 14% lesser compared to EAPS-E, whereas, the transaction duration of EAPS-L is 18% higher than EAPS-E. EAPS-M balances the trade off between energy consumption and transaction duration.

6 Discussion

Wake-up radio. WUR mechanisms such as 802.11ba [8], [9] can be used to enhance EAPS in several ways. For example, as soon as a station finishes sending its uplink packet(s), the primary radio can switch into sleep mode, and the station will receive the schedule message via its low-power WUR. The primary radio will then wake up at the scheduled time to receive the downlink packet. To further reduce the idle energy caused by prediction inaccuracy, the WUR can be scheduled to wake up at $\delta' - 2\sigma$ and wait for a command to wake up the primary radio. As another example, once the downlink packet arrives on the wired interface of the AP, the AP uses EAPS to compute the packet delivery delay. Assuming that the wake-up delay of the primary radio is β [8], the AP sends the wake-up packet at $\delta' - \beta$ to make sure the station's primary radio will be awake on time for downlink delivery.

Mesh networks. As discussed in §3.3, the primary types of networks used in this paper are smart home environments where an AP is connected to an Internet modem, and campus and business deployments where APs communicate via an Ethernet infrastructure. EAPS can also be used in mesh deployments. In this case, the backbone communication between APs (mesh nodes) introduces a wireless-to-wireless switching delay. This delay primarily depends on the bandwidth difference between the backbone link (AP-AP) and access links (AP-station). For example, assume a 160 MHz channel (in the 5 GHz band) is used to form the backbone, while each AP operates on a 20 MHz or 40 MHz channel (in the 2.4 GHz band). This configuration is prevalent because most of the existing IoT stations operate in the 2.4 GHz band, and WiFi mesh systems are usually tri-band and dedicate a channel (in the 5 GHz band) to their backbone. With this configuration, the delay caused by the backbone would be negligible and a method similar to that mentioned in §3.3 can be used to measure the delay from each AP to the server. If backbone links suffer from congestion and significant interference, EAPS can be used to predict packet switching delay over the backbone. As an alternative, more efficient strategy, EAPS could run on a central machine and allow the stations to receive their downlink packet from the AP offering the lowest delay. We leave these enhancements as future works.

Computation offloading. If the AP is not powerful enough to train the model, the training could be offloaded to a cloud or fog computing platform. In any case, edge computing is essential to perform scheduling immediately and convey the sleep schedule to the station.

7 Conclusion

In this paper, we presented the design, implementation, and evaluation of a predictive scheduling mechanism, named

EAPS, which allows IoT stations to transition to sleep mode and wake up when their downlink packet(s) is expected to be delivered. The proposed solution benefits from edge computing, meaning that sleep scheduling is performed at the network edge and by the AP. We presented an AP architecture capable of collecting queues status, channel condition, and packet transmission and reception instances. Once the AP receives an uplink packet, a machine learning model is used to compute the sleep delay, and the station is informed about its schedule using a high-priority data plane. Using empirical evaluations, we confirmed the significant enhancement of EAPS in terms of energy efficiency and transaction delay.

EAPS can be used to augment the power saving mechanisms of 802.11 such as APSD and TWT (introduced in 802.11ah and 802.11ax). The next generation of IoT stations that support TWT can set up their wake up time based on the sleep schedule computed by AP. By protecting IoT stations against the effect of concurrent traffic and interference, EAPS is a particularly useful method in scenarios where both regular and IoT stations exist. EAPS can lower the energy cost of households and reduce the impact of IoT on global ICT energy footprint.

REFERENCES

[1] Cypress Semiconductor. CYW43907: IEEE 802.11 a/b/g/n SoC with an Embedded Applications Processor. [Online]. Available: http://www.cypress.com/file/298236/download

[2] AVNET Inc. BCM4343W: 802.11b/g/n WLAN, Bluetooth and BLE SoC Module. [Online]. Available: https://products.avnet.com/opasdata/d120001/medias/docus/138/AES-BCM4343W-M1-G_data_sheet_v2_3.pdf

[3] S. Tozlu, M. Senel, W. Mao, and A. Keshavarzian, "Wi-Fi enabled sensors for internet of things: A practical approach," *IEEE Communications Magazine*, vol. 50, no. 6, pp. 134–143, 2012.

[4] "Analysis of the impact of background traffic on the performance of 802.11 power saving mechanism," *IEEE Communications Letters*, vol. 13, no. 3, pp. 164–166, 2009.

[5] J. Manweiler and R. Roy Choudhury, "Avoiding the rush hours: WiFi energy management via traffic isolation," in *MobiSys*, 2011, pp. 253–266.

[6] S. Y. Jang, B. Shin, and D. Lee, "An adaptive tail time adjustment scheme based on inter-packet arrival time for IEEE 802.11 WLAN," in *ICC*. IEEE, 2016, pp. 1–6.

[7] A. Vinhas, V. Bernardo, M. Pascoal Curado, and T. Braun, "Performance analysis and comparison between legacy-PSM and U-APSD," *CRC*, pp. 1–12, 2013.

[8] D.-J. Deng, M. Gan, Y.-C. Guo, J. Yu, Y.-P. Lin, S.-Y. Lien, and K.-C. Chen, "IEEE 802.11 ba: Low-power wake-up radio for green IoT," *IEEE Communications Magazine*, vol. 57, no. 7, pp. 106–112, 2019.

[9] D. Bankov, E. Khorov, A. Lyakhov, and E. Stepanova, "IEEE 802.11 ba—Extremely Low Power Wi-Fi for Massive Internet of Things—Challenges, Open Issues, Performance Evaluation," in *BlackSeaCom*. IEEE, 2019, pp. 1–5.

[10] E. Rozner, V. Navda, R. Ramjee, and S. Rayanchu, "NAPman: Network-assisted power management for WiFi devices," in *MobiSys*. ACM, 2010, pp. 91–106.

[11] A. J. Pyles, X. Qi, G. Zhou, M. Keally, and X. Liu, "SAPSM: Smart adaptive 802.11 PSM for smartphones," in *UbiComp*. ACM, 2012, pp. 11–20.

[12] S. Sundaresan, N. Magharei, N. Feamster, R. Teixeira, and S. Crawford, "Web performance bottlenecks in broadband access networks," *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 1, pp. 383–384, 2013.

[13] C. Pei, Y. Zhao, G. Chen, R. Tang, Y. Meng, M. Ma, K. Ling, and D. Pei, "WiFi can be the weakest link of round trip network latency in the wild," in *INFOCOM*. IEEE, 2016, pp. 1–9.

[14] T. Høiland-Jørgensen, P. Hurtig, and A. Brunstrom, "The good, the bad and the WiFi: Modern AQMs in a residential setting," *Computer Networks*, vol. 89, pp. 90–106, 2015.

[15] 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*, 2017, pp. 139–151.

[16] A. Showail, K. Jamshaid, and B. Shihada, "Buffer sizing in wireless networks: challenges, solutions, and opportunities," *IEEE Communications Magazine*, vol. 54, no. 4, pp. 130–137, 2016.

[17] M. Heusse, F. Rousseau, G. Berger-Sabbatel, and A. Duda, "Performance anomaly of 802.11b," in *INFOCOM*, vol. 2. IEEE, 2003, pp. 836–843.

[18] B. Peck and D. Qiao, "A practical PSM scheme for varying server delay," *IEEE Transactions on Vehicular Technology*, vol. 64, no. 1, pp. 303–314, 2015.

[19] K. Sui, M. Zhou, D. Liu, M. Ma, D. Pei, Y. Zhao, Z. Li, and T. Moscibroda, "Characterizing and improving WiFi latency in large-scale operational networks," in *MobiSys*, 2016, pp. 347–360.

[20] K.-Y. Jang, S. Hao, A. Sheth, and R. Govindan, "Snooze: Energy management in 802.11n WLANs," in *Co-NEXT*. ACM, 2011, p. 12.

[21] J. Sheth and B. Dezfouli, "Enhancing the energy-efficiency and timeliness of IoT communication in WiFi networks," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 9085–9097, 2019.

[22] L. Liu, X. Cao, Y. Cheng, and Z. Niu, "Energy-efficient sleep scheduling for delay-constrained applications over WLANs," *IEEE Transactions on Vehicular Technology*, vol. 63, no. 5, pp. 2048–2058, 2014.

[23] F. Wilhelmi, S. Barrachina-Munoz, B. Bellalta, C. Cano, A. Jonsson, and V. Ram, "A flexible machine-learning-aware architecture for future WLANs," *IEEE Communications Magazine*, vol. 58, no. 3, pp. 25–31, 2020.

[24] J. Malinen. hostapd: IEEE 802.11 AP, IEEE 802.1X/WPA/WPA2/EAP/RADIUS Authenticator. [Online]. Available: https://w1.fi/hostapd/

[25] Y.-H. Wei, Q. Leng, S. Han, A. K. Mok, W. Zhang, and M. Tomizuka, "RT-WiFi: Real-time high-speed communication protocol for wireless cyber-physical control applications," in *RTSS*. IEEE, 2013, pp. 140–149.

[26] C. Powell, C. Desiniotis, and B. Dezfouli, "The fog development kit: A development platform for SDN-based edge-fog systems," *IEEE Internet of Things Journal*, 2019.

[27] V. Jacobson, "Congestion avoidance and control," in *SIGCOMM computer communication review*, vol. 18, no. 4. ACM, 1988, pp. 314–329.

[28] R. Ludwig and K. Sklower, "The Eifel retransmission timer," *SIGCOMM Computer Communication Review*, vol. 30, no. 3, pp. 17–27, 2000.

[29] R. Russell and H. Welte. Linux netfilter hacking howto. [Online]. Available: http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO

[30] K.-c. Lan and J. Heidemann, "A measurement study of correlations of internet flow characteristics," *Computer Networks*, vol. 50, no. 1, pp. 46–62, 2006.

[31] Y. Xiao, Y. Cui, P. Savolainen, M. Siekkinen, A. Wang, L. Yang, A. Ylä-Jääski, and S. Tarkoma, "Modeling energy consumption of data transmission over Wi-Fi," *IEEE Transactions on Mobile Computing*, vol. 13, no. 8, pp. 1760–1773, 2013.

[32] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[33] K. M. Koudjonou and M. Rout, "A stateless deep learning framework to predict net asset value," *Neural Computing and Applications*, pp. 1–19, 2019.

[34] F. Chollet *et al.*, "Keras, github," *GitHub repository, https://github.com/fchollet/keras*, 2015.

[35] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[36] 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.