Santa Clara University
Scholar Commons

Engineering Ph.D. Theses Student Scholarship

6-9-2022

Enhancing the Quality of Service and Energy Efficiency of WiFi-based IoT Networks

Jaykumar Sheth

Follow this and additional works at: https://scholarcommons.scu.edu/eng_phd_theses

Part of the Computer Engineering Commons

Santa Clara University

Department of Computer Science and Engineering

Date: June 9, 2022

I HEREBY RECOMMEND THAT THE THESIS

PREPARED UNDER MY SUPERVISION BY

Jaykumar Sheth

ENTITLED

Enhancing the Quality of Service and Energy Efficiency of WiFi-based IoT Networks

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE AND ENGINEERING

Behnam Dezfouli
Behnam Dezfouli (Jun 11, 2022 17:13 PDT)

Thesis Advisor
Dr. Behnam Dezfouli

N. Ling
N. Ling (Jun 13, 2022 15:15 PDT)

Chairman of Department
Dr. Nam Ling

Thesis Reader
Dr. Silvia Figueira

Thesis Reader
Dr. Tokunbo Ogunfunmi

Thesis Reader
Dr. Sundar Sankaran

Thesis Reader
Dr. Yuhong Liu

Enhancing the Quality of Service and Energy Efficiency of WiFi-based IoT Networks

By

Jaykumar Sheth

Dissertation

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Science and Engineering
in the School of Engineering at
Santa Clara University, 2022

Santa Clara, California

Acknowledgments

First and foremost, I would like to express my gratitude to my advisor Dr. Behnam Dezfouli. Thank you for being patient and guiding me through each stage of research, right from preliminary brainstorming till addressing reviewers' comments. He has always been accessible and his spontaneous feedback has provided me with frequent chances to improve. Dr. Dezfouli has always prioritized my overall development, has looked out for me, and supported me in many ways beyond research. His unwavering support and belief in me enabled me to find my strength and weaknesses, and work towards becoming a better researcher. One of his teachings that has been pivotal during my transition from an engineer to a researcher is that purposeful research is not only about coming up with novel ideas, but also equally important is to scientifically validate the ideas and present them in a methodical manner, such that the research community can easily understand, adopt, and develop new methods based on your findings. Dr. Dezfouli is an inspiration and a role model to me, and I look forward to continue learning from him.

I would also like to thank Dr. Silvia Figueira, Dr. Tokunbo Ogunfunmi, Dr. Yuhong Liu, and Dr. Sundar Sankaran for being Ph.D. committee members and for their valuable suggestions and assistance in enhancing the quality of my research.

A special thanks to the open-source community and engineering enthusiasts around the world. Directly and indirectly they have made such an impact on my research. I plan to continue giving back to the community. Especially, Toke Hoiland-Jorgensen's Ph.D. work on Bufferbloat was the key inspiration for my projects presented

in this thesis. Brendan Greg's presentations and literature on performance engineering inspired me to learn eBPF technology and employ it in my projects. Furthermore, thank you JVR Murthy and Arista Networks for providing an opportunity to work on next generation WiFi systems and tackling the challenges faced in production environment.

Friends, lab mates, colleagues inside and outside the lab have made this journey enjoyable. Vikram Ramanna, your support while working on the papers and conducting experiments was instrumental. Dr. Amir Dezfouli, Cyrus Miremadi, Alex Paiz, and An Vu, thank you for collaborating on various projects, your input along with technical discussions significantly helped in enhancing the impact of the projects presented in this thesis. I would also like to thank my closest friends, Om Mer and Hem Zaveri, for standing by me through happy and challenging times.

I am forever indebted to my parents who worked tirelessly over all these years to make sure I had access to every opportunity. My parents have always been the embodiment of how pure love and selfless support can reach across the breadth of the world's oceans and given me strengths and comfort throughout this journey. Additionally, my sister — Kruti Sheth's support has been vital since childhood. I would also like to thank my brother Dr. Vishal Mehta for paving the path to pursue research. He has been there with me since I stepped in the USA and he has been a mentor in many ways. I would also like to thank my in-laws for their patience and constant support through this journey.

Last but not the least, I am utmost grateful to my wife Brinda. Without her love and constant support, I would not have been able to achieve my current standing. She has been my fiercest advocate and ardent critic, which has been instrumental in honing my skills and enhancing the quality of my research. This thesis, and my life, are as much hers as they are mine. All the sacrifices that you have made on my behalf for pursuing our dreams have made this feat possible.

Dedicated to the loving memory of my mother

Neeta Sheth

All of this would not have been possible without your unbounded belief in me.

Enhancing the Quality of Service and Energy Efficiency of WiFi-based IoT Networks

Jaykumar Sheth

Department of Computer Science and Engineering
Santa Clara University
Santa Clara, California
2022

ABSTRACT

The 802.11 standard, known as WiFi, is currently being used for a wide variety of applications including Internet of Things (IoT). However, the contention between the traffic of IoT stations (STAs) as well as the contention between these flows and regular user-generated traffic reduces the energy efficiency and timeliness of IoT communication. To remedy this problem, in this thesis, we take the following approaches for mitigating the challenges faced by WiFi-based IoT networks: First, we highlight the importance of observability with respect to WiFi networks and how it helps the researchers to better examine the dynamics of issues and its causes. We then develop two tools that enable high-rate monitoring of the Linux networking stack. These tools rely on the fact that all data traffic in WiFi networks flows through the Access Point (AP). This enables us to deploy these tools on only the APs and not each connected device; thus enabling monitoring of large-scale networks. Second, we enhance this tool by utilizing the extended Berkeley Packet Filter (eBPF) technology for monitoring of the networks without modifying any kernel modules to analyze the delays incurred by the packets at different parts of the networking subsystem on the APs and also monitor the energy consumption of the associated STAs. Third, utilizing these tools, we obtain insights and measurements to design a scheduling algorithm that computes per-packet priorities

to arbitrate the contention between the transmission of IoT packets. This algorithm employs a least-laxity first (LLF) scheme that assigns priorities based on the remaining wake-up time of the destination STAs. Fourth, we estimate the interval uplink-request and downlink-response due to overheads in the wired and wireless networking components across the path of the packet from the edge device and server. We facilitate the STA with the estimated wired and wireless transit delay, such that the STA can utilize this information to transition to low power sleep state during the packet's transit; thus, enhancing the STAs energy efficiency. Fifth, focusing on the power-save functionality introduced in next-generation WiFi standard, known as Target Wake-up Time (TWT), we first, highlight the importance of traffic characterization and shortcomings of existing methods. We then propose a transport layer-based traffic characterization method that can accurately capture inter-packet and inter-burst intervals on a per-flow basis in the presence of factors such as channel access and packet preparation delay. By addressing the challenges due to the shared nature of WiFi spectrum the contributions in this thesis provide open-source tools for better understanding the internals of networking stack and methods for improving the energy efficiency and quality of service of WiFi communication in IoT networks.

Table of Contents

1 Introduction **1**

1.1 Overview 1

1.2 Problem Statement 2

1.2.1 First Research Problem: Observability and Monitoring of Linux Networking Stack 2

1.2.2 Second Research Problem: Linux Networking Monitoring in an non-intrusive manner 3

1.2.3 Third Research Problem: IoT Traffic Prioritization 4

1.2.4 Fourth Research Problem: Predictive Sleep Scheduling of IoT Stations 5

1.2.5 Fifth Research Problem: Traffic Characterization for Efficient TWT Scheduling in 802.11ax IoT Networks 6

1.3 Contributions 7

1.4 Scope 9

1.5 Thesis Organization 11

2 Literature Review **13**

2.1 Introduction 13

2.2 Power Saving Optimizations in WiFi Networks 13

2.2.1 Power Saving Methods in 802.11 (a/b/g/n/ac) Standards 13

2.2.2 Power Saving Methods 802.11 (ax) Standard 15

2.3 Packet Scheduling 17

2.3.1 Traffic Prioritization 17

2.3.2 RTT and Delay Estimation 19

2.4 Linux Network Stack Monitoring 22

2.4.1 Sniffer-based Approaches 22

2.4.2 eBPF (enhanced Berkeley Packet Filter) 23
2.4.3 Utilizing Linux Tools 24

3 MonFi: A Tool for High-Rate, Efficient, and Programmable Monitoring of WiFi Devices 27
3.1 Introduction 27
3.2 Existing Monitoring Tools 28
3.3 Design and Implementation of MonFi 30
3.3.1 Architecture 30
3.3.2 Extensive Data Collection Across the Protocol Stack 32
3.3.2.1 PHY and MAC measurements 32
3.3.2.2 Monitoring qdisc 35
3.3.3 Monitoring the Host system 35
3.3.3.1 Dedicating computing resources to MonFi 36
3.3.4 Sharing the Collected Data with User-space 37
3.3.4.1 Event-based data collection (EDC) 37
3.3.4.2 Polling-based data collection (PDC) 38
3.3.4.3 Event and Polling-based Data Collection (EPDC) 38
3.4 Performance Evaluation 39
3.4.0.1 MonFI vs ioctl and debugfs 39
3.4.0.2 Impact of processor load on monitoring performance . . 40
3.4.0.3 Impact of packet switching on monitoring performance . 42
3.5 Summary 43

4 FLIP: A Framework for Leveraging eBPF to Augment WiFi Access Points and Investigate Network Performance 45
4.1 Introduction 45
4.2 System Architecture 47
4.2.1 AP's Networking Stack 48
4.2.2 Leveraging eBPF for Collecting Monitoring Data from the Kernel 48
4.3 Delay Analysis of Switching Packets from the Wired Interface to the Wireless Interface 50

4.3.1 Power Saving Methods of 802.11 50
4.3.2 FLIP's Methodology for Monitoring Wired-to-wireless Switching Path 51
4.3.2.1 Queuing Disciplines (qdisc) 51
4.3.2.2 mac80211 53
4.3.2.3 Driver and WL-NIC 53
4.3.3 Empirical Evaluation of Wired-to-wireless Switching Delay 54
4.4 Passive Monitoring of Stations' Energy Consumption 58
4.4.1 FLIP's Methodology for Monitoring the Energy Consumption of Stations 58
4.4.2 Empirical Evaluation of the Accuracy of Passive Energy Monitoring 60
4.5 Summary 62

5 Enhancing the Energy-Efficiency and Timeliness of IoT Communication in WiFi Networks 65
5.1 Introduction 65
5.2 System Overview 67
5.2.1 Traffic Prioritization 67
5.2.2 System Model 68
5.3 Scheduling Mechanism 68
5.3.1 Acceleration Eligibility 69
5.3.2 Queue Configuration 71
5.3.3 Enqueue Algorithm 74
5.4 Implementation 76
5.4.1 WiFi Logger Module (WiLog) 78
5.4.2 Scheduler Module 78
5.4.3 qdisc Module 79
5.5 Simulation Results 80
5.6 Empirical Evaluation 83
5.6.1 Testbed 83
5.6.1.1 Hardware 83

5.6.1.2 Publish/subscribe model 84
5.6.1.3 Background traffic 84
5.6.1.4 Energy measurement platform 85
5.6.2 Methodology 86
5.6.3 Results and discussions 87
5.7 Summary 89

6 EAPS: Edge-Assisted Predictive Sleep Scheduling for 802.11 IoT Stations . 91
6.1 Introduction 91
6.2 Delay Analysis and AP Development 92
6.2.1 Delay Components 92
6.2.2 AP Development 93
6.2.3 Communication Delay Between AP and Server 95
6.2.4 AP to Station Delivery Delay 96
6.2.4.1 Input traffic rate through wired interface 97
6.2.4.2 qdisc queues 98
6.2.4.3 Wireless channel condition 98
6.2.4.4 Driver's transmission queues 99
6.2.4.5 Summary of the features collected 100
6.2.5 Schedule Announcement 101
6.3 Predictive Scheduling 102
6.3.1 Traffic Generation 102
6.3.2 Data Collection 105
6.3.3 Data Pre-processing 105
6.3.4 Regression Models 106
6.3.5 Model Evaluation 109
6.4 Empirical Evaluation 113
6.4.1 Testbed 113
6.4.2 Baselines and EAPS Variations 114
6.4.3 Results 116

6.5 Discussion 118
6.6 Summary 119

7 Traffic Characterization for Efficient TWT Scheduling in 802.11ax IoT Networks 121
7.1 Introduction 121
7.2 Traffic Pattern Analysis 122
7.3 Traffic Characterization via Channel Utilization, BSR, and Packet Sniffing 126
7.3.1 Channel Utilization (CU) 126
7.3.2 Packet Sniffing 126
7.3.3 Buffer Status Report (BSR) 127
7.4 Source-assisted Traffic Characterization 129
7.4.1 Design and Implementation 129
7.4.1.1 Packet Modification in the MAC or NIC 130
7.4.1.2 Packet Modification in the TCP Layer 130
7.4.2 Analytical Comparison 132
7.4.3 Empirical Evaluations of SATRAC 133
7.4.4 Sample TWT Allocation Scenario 135
7.5 Summary 136

8 Conclusion and Future Work 137

References 140

Notations 156

Glossary 157

This page is intentionally left blank

List of Figures

2.1 Literature review overview 14

3.1 The architecture of MonFi 31
3.2 Measurement collection rate (per second) 40
3.3 Evaluating MonFi's performance 41
3.4 MonFi's performance before and after CPU dedication 42

4.1 The FLIP architecture 47
4.2 Delay components 52
4.3 Delay components analysis via FLIP 54
4.4 Evaluating efficiency of FLIP's passive energy monitoring 61
4.5 Energy consumption measurement via FLIP 62

5.1 Illustration of proposed scheduling algorithm 70
5.2 Implementation architecture of the proposed Wiotap AP 78
5.3 Performance evaluation of WIOTAP w.r.t. number of IoT stations 81
5.4 Performance evaluation of WIOTAP w.r.t. number of IoT stations and regular traffic 81
5.5 Performance evaluation of WIOTAP w.r.t. amount of regular traffic . . . 81
5.6 Evaluating scalibility of WIOTAP 83
5.7 WIOTAP testbed architecture 84
5.8 Performance of WIOTAP in edge computing scenario 87
5.9 Performance of WIOTAP in cloud computing scenario 88

6.1 Delay components between a station and a server 93
6.2 AP architecture 94
6.3 Traffic characterisation 103

6.4 Comparison of generated traffic with real world scenarios 106

6.5 Mean Absolute Error (MAE) of machine learning algorithms versus the length of training set 109

6.6 MAE of machine learning algorithms with respect to feature history . . . 110

6.7 MAE of machine learning algorithms versus transaction's AC 111

6.8 ECDF of prediction errors 112

6.9 Effect of transaction history on MAE of Long Short-Term Memory (LSTM) 113

6.10 Processing time of the prediction algorithms 113

6.11 CDF of prediction error 115

6.12 Performance comparison of EAPS with power saving mechanisms (ND) . 116

6.13 Performance comparison of EAPS with power saving mechanisms (HD) . 116

7.1 Micro-burst and macro-burst characterization. 123

7.2 Inter-packet intervals for the Sensor scenario 124

7.3 Inter-packet intervals for the Camera scenario 124

7.4 Inter-packet intervals for the Video Streaming scenario. 125

7.5 Packets per second 125

7.6 Traffic characteristics and Buffer Status Report (BSR) 128

7.7 Traffic characteristics and data generation instances 133

7.8 Empirical comparison of SATRAC 134

7.9 Sequential and overlapping TWT allocations 135

List of Publications

Journals:

(i) Jaykumar Sheth and Behnam Dezfouli. Enhancing the Energy-Efficiency and Timeliness of IoT Communication in WiFi Networks. *IEEE Internet of Things Journal*, 6(5):9085–9097, 2019.

(ii) Jaykumar Sheth, Cyrus Miremadi, Amir Dezfouli, and Behnam Dezfouli. EAPS: Edge-Assisted Predictive Sleep Scheduling for 802.11 IoT Stations. *IEEE Systems Journal*, 16(1):591–602, 2021.

(iii) Behnam Dezfouli, Vahid Esmaeelzadeh, Jaykumar Sheth, and Marjan Radi. A Review of Software-defined WLANs: Architectures and Central Control Mechanisms. *IEEE Communications Surveys and Tutorials*, 21(1):431–463, 2018.

(iv) Vikram Ramanna, Jaykumar Sheth, Simon Liu, and Behnam Dezfouli . Towards Understanding and Enhancing Association and Long Sleep in Low-power WiFi IoT Systems. *IEEE Transactions on Green Communications and Networking*, 5.4, 1833-1845, 2021

Conferences:

(i) Jaykumar Sheth and Behnam Dezfouli. MonFi: A Tool for High-Rate, Efficient, and Programmable Monitoring of WiFi Devices. In *IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–7. IEEE, 2021.

(ii) Jaykumar Sheth, Vikram Ramanna, and Behnam 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*, pages 117–125, 2021.

(iii) Jaykumar Sheth, Vikram Ramanna, and Behnam Dezfouli. Traffic Characterization for Efficient TWT Scheduling in 802.11ax IoT Networks. Pending Acceptance in *Proceedings of IEEE Global Communications Conference (GLOBECOM)*, 2022

(iv) Aastha Chawla, Nidusha Kannan, Sreya Goyalia, Vikram Ramanna, Jaykumar Sheth, and Behnam Dezfouli. SEMFI: A Software-Based and Real-Time Energy Monitoring Platform for WiFi IoT Devices. Pending Acceptance. *IEEE Global Humanitarian Technology Conference (GHTC)*, 2022.

Chapter 1

Introduction

1.1 Overview

The Internet of Things (IoT) is the enabler of applications such as remote medical monitoring, building automation, industrial automation, and smart cities. It is estimated that there will be about 14.7 billion Internet of Things (IoT) connected devices in 2023, up from 6.1 billion in 2018 [1]. To offer ease of deployment and support mobility, most of these applications rely on wireless communication. Various wireless technologies, such as 802.15.4, NB-IoT, LoRa, LTE, and 802.11, are currently being used to connect these devices [2]. Meanwhile, 802.11 is particularly important due to several reasons: First, 802.11 networks are widely deployed in home, enterprise, and commercial environments. Therefore, for example, 802.11-based smart home systems do not require the installation of additional wireless infrastructure. Second, the power consumption of 802.11 transceivers has been significantly reduced during the recent years. This has been achieved by both new chip manufacturing technologies as well as the various power saving methods proposed for this standard [3, 4, 5, 6, 7, 8, 9]. Third, compared with cellular networks, 802.11 operates in unlicensed bands and thereby, its usage is free of charge. Finally, compared to 802.15 technologies, 802.11 offers considerably higher data rates, which enable the usage of this standard in applications such as medical monitoring and industrial control [10, 11]. The higher data rate also enables the radio transceiver to finish its transmission faster and switch to a low-power mode.

1.2 Problem Statement

Building a WiFi network catering to the needs of efficient IoT deployments imposes major challenges that can be broadly categorized into the following: (i) *Energy Efficiency*: appropriate energy should be utilized for WiFi communication of the battery-powered and energy-constrained IoT devices, (ii) *Observability*: power consumption patterns and the efficiency of last-hop delivery should be monitored to develop novel and efficient algorithms, (iii) *Timeliness*: adequate packet prioritization and scheduling techniques should be developed for orderly delivery in presence of contending traffic from other WiFi stations (STAs) in the network. This section elaborates on the research problems of this thesis.

1.2.1 First Research Problem: Observability and Monitoring of Linux Networking Stack

Collecting monitoring data from WiFi devices enables researchers to study network operation [12, 13, 14, 15], improve performance [16, 17, 18], and secure these networks [19, 20, 21, 22]. For example, methods have been proposed to manage channel assignment to Access Points (APs), control the association of stations, and adjust transmission power, to mention a few [23, 24, 11]. Although collecting monitoring data every few seconds would be enough for some algorithms (e.g., channel assignment), immediate reactions to network dynamics require high-rate, real-time monitoring. A sample scenario is per-flow and per-packet scheduling methods in dense networks [18]. Also, existing research shows the importance of packet-level analysis for power management [25, 16]. We also observe the adoption of data-driven and machine-learning-based methods for performance enhancement (e.g., delay reduction [14], power management [26, 27, 28]) and security provisioning [19, 20, 21, 22].

RESEARCH QUESTION 1.2.1.1 How to design and develop high-rate, real-time, efficient, and programmable monitoring tool to collect measurements pertaining to the efficiency of WiFi communication between APs and stations?

1.2.2 Second Research Problem: Linux Networking Monitoring in an non-intrusive manner

The WiFi networking stack is complex and includes multiple layers across the Wireless Network Interface Card (WL-NIC), driver, Linux kernel modules, and user-space daemons. Although there exist tools that provide visibility into some of these layers, the performance and range of visibility of these tools are far from what is needed to analyze these networks and design solutions for performance enhancement effectively. Due to this shortcoming, a large number of existing works rely on simulation. Also, when high-rate monitoring is necessary, existing works rely on packet capture and static data analysis [19, 20, 21]. Another category of works relies on tools that have been primarily designed for infrequent monitoring and configuration [13, 14]. For example, Linux-based tools such as `iw` and `ethtool` can be used to collect *some* of the operational data from the WiFi stack; however, the sampling rate and efficiency of these tools are far below what is required for high-rate monitoring. However, these techniques do provide any flexibility extension to their implemented logic requires modification of the tool or the kernel. This process is cumbersome and less secure, as any programmatic errors can lead to failure of whole module or the entire kernel.

RESEARCH QUESTION 1.2.2.1 How to monitor the networking stack without requiring any modifications to the kernel components or modules?

RESEARCH QUESTION 1.2.2.2 Can the delays incurred at networking stack and the

energy consumption of the STA's WiFi Network Interface Card (NIC) be analyzed from the AP without utilizing additional hardware?

1.2.3 Third Research Problem: IoT Traffic Prioritization

From the timeliness point of view, 802.11 differentiates between real-time (e.g., voice and video) and elastic (e.g., email, file transfer) flows by defining *voice*, *video*, *best-effort*, and *background* access categories. However, these access categories only accommodate the four traffic categories of regular (non-IoT) user devices and cannot differentiate the existence of IoT traffic. Therefore, IoT stations suffer from longer delays and also waste their energy because of idle listening before their downlink packets are delivered. Due to the benefits entailed by the Adaptive PSM (APSM) (cf. 2.2.1), we observe that the state-of-the-art low-power 802.11 transceivers support this mode [29, 30, 6]. However, the tail time duration might also result in a higher energy consumption if the access point's downlink traffic is high and packet transmissions are not properly scheduled based on energy efficiency concerns. In other words, when the number of regular or IoT stations increases, the amount of time spent in the tail time increases without positively affecting timeliness or energy efficiency.

RESEARCH QUESTION 1.2.3.1 How to design and develop a scheduling algorithm that prioritizes IoT traffic based on the remaining tail time of IoT STAs, such that the Quality of Service (QoS) and energy efficiency of the WiFi communication in IoT networks is improved?

1.2.4 Fourth Research Problem: Predictive Sleep Scheduling of IoT Stations

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 four options with regards to power saving methods (cf. 2.2.1) 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 Automatic Power Save Delivery (APSD) or Target Wake-up Time (TWT) and periodically check if the downlink packet has arrived,

Case (i) minimizes delay but does not offer any power efficiency. Case (ii) causes long end-to-end delays [31, 28] 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.

[1]Considering user interactions, studies show that every 10 ms increase in network access results in a 1000 ms increase in page load time [32].

Therefore, none of these cases are suitable for applications where both delay and energy efficiency are the essential performance metrics.

RESEARCH QUESTION 1.2.4.1 How to estimate the total wired and wireless transit delay between Uplink (UL) and Downlink (DL) and adjust the sleep schedule of the STAs to maximize energy efficiency and minimize the additional delay?

1.2.5 Fifth Research Problem: Traffic Characterization for Efficient TWT Scheduling in 802.11ax IoT Networks

The newly-introduced 802.11ax standard provides a method called TWT for assigning *service periods* to STAs. Compared to the earlier power-saving modes, TWT allows for potentially higher energy efficiency and throughput. Specifically, by properly assigning service periods to STAs, channel contention reduces, packet buffering delay in the AP drops, and packet aggregation efficiency enhances [33]. Nevertheless, to realize the benefits of TWT scheduling, accurate characterization of the traffic flows of STAs is required by an AP to allocate service periods that address STAs' traffic requirements [34, 33, 35]. To this end, the most-widely used methods are Channel Utilization (CU) estimation, packet sniffing, and Buffer Status Report (BSR) [36, 37]. However, as we will show in this thesis, none of these methods provide high accuracy, especially in the presence of channel access delay. Another approach is to gradually and dynamically adjust TWT assignments over time, based on STAs' demands.

RESEARCH QUESTION 1.2.5.1 How accurately characterize traffic of the associated STAs for allocating TWT service periods?

1.3 Contributions

To achieve the aforementioned objectives, this thesis presents the following contributions:

(i) We propose *MonFi*, a publicly-available, open-source tool for high-rate, efficient, and programmable monitoring of the WiFi communication stack. With this tool, regular user-space applications can specify their required measurement parameters, monitoring rate, and measurement collection method as event-based, polling-based, or a hybrid of both. We also propose methods to ensure deterministic sampling rate, regardless of the processor load caused by other processes including packet switching. In terms of sampling rate and processing efficiency, we show that MonFi outperforms the Linux tools used to monitor the communication stack.

(ii) We propose *FLIP*, a framework for leveraging eBPF (Enhanced-Berkeley Packet Filter) to augment access points and investigate the performance of WiFi networks. Using this framework, we focus on two important aspects of monitoring the WiFi stack. First, considering the high delay experienced by packets at access points, we show how switching packets from the wired interface to the wireless interface can be monitored and timestamped accurately at each step. By relying on the FLIP framework, we build a testbed and investigate the factors affecting packet delay experienced at access points. Second, we present a novel approach that allows access points to track the duty-cycling pattern and energy consumption of their associated stations accurately and without the need for any external energy measurement tools. We validate the high energy measurement accuracy of FLIP by empirical experiments and comparisons against a commercial tool.

(iii) We propose *Wiotap*, an enhanced WiFi access point that includes a downlink packet scheduling algorithm. In addition to assigning higher priority to IoT traffic

compared to regular traffic, the scheduling algorithm computes per-packet priorities to arbitrate the contention between the transmission of IoT packets. This algorithm employs a least-laxity first (LLF) scheme that assigns priorities based on the remaining wake-up time of the destination stations. We confirm the accuracy of WIOTAP via empirical evaluation and we utilize simulation to show the scalability of the proposed system.

(iv) We propose *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.

(v) Finally, we empirically study and reveal that the existing methods (i.e., channel utilization estimation, packet sniffing, and buffer status report) do not provide adequate accuracy. To remedy this problem, we propose a traffic characterization method that can accurately capture inter-packet and inter-burst intervals on a per-flow basis in the presence of factors such as channel access and packet preparation delay. We empirically evaluate the proposed method and confirm its superior traffic characterization performance against the existing ones. We also present a sample TWT allocation scenario that leverages the proposed method to enhance throughput.

1.4 Scope

The following clarify the scope of this thesis:

(i) The proposed methods comply by the 802.11 standard specifications. Hence, these methods can be applied to any application domain that utilizes WiFi-based communication, such as process automation, factory automation, building automation systems, and intra-vehicle communication.

(ii) This thesis is based on three assumptions: First, wireless communication happens over a single hop, i.e., between the edge-devices and AP. Second, as WiFi communication utilizes the unlicensed spectrum, high interference will result due to background traffic of STAs operating on the same channel. Third, as IoT devices are energy constrained and IoT traffic intensity is relative lower compared to other WiFi devices, it is very important to prioritize the IoT traffic and optimize the energy efficiency of IoT STAs.

(iii) Most commercial WiFi adapters rely on a SoftMAC architecture [38], especially considering the ease of updating. The WiFi drivers utilized in this thesis are SoftMAC drivers. SoftMAC drivers implement a part of the MAC layer management entity (MLME) in software, thereby utilizing the host system's processing resources. Only time-critical MAC functions, such as managing timeouts, inter-frame spacing, and channel access backoff, are implemented in the hardware. The other type of drivers, referred as FullMAC utilizes an additional processor on the NIC to implement all MAC functions within the firmware.

(iv) As all traffic passes through the AP, this thesis considers the AP to be an appropriate entity for monitoring the WiFi communication. On the contrary, monitoring each associated STA via physically attaching probes is not a scalable approach.

(v) This thesis does not propose any channel access arbitration mechanism for real-time packet delivery.

(vi) The methods presented in thesis do not alter any part of the PHY and MAC layer parameters, such the channel contention parameters mentioned in the 802.11 standard specifications.

(vii) The proposed methods in this thesis uses Qualcomm® Atheros-based WiFi devices, capable of operating in both STA mode and AP mode utilizing hostapd. This devices use Linux as there operating system. Additionally, targeting resource-constrained devices, two widely-used Real-Time Operating Systems (RTOSs): FreeRTOS [39, 40] and ThreadX [41, 42] are utilized as the STA's operating system.

1.5 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 provides the background of this research, and reviews existing literature with respect to (i) WiFi networking stack monitoring, (ii) power saving techniques used in WiFi networks, and (iii) packet scheduling methods for enhancing the performance of WiFi-based IoT devices. Chapter 3 presents MonFi, an open-source tool for high-rate, efficient, and programmable monitoring of the WiFi communication stack. Chapter 4 presents FLIP, a framework for leveraging eBPF to augment access points and investigate the performance of WiFi networks. Chapter 5 presents WIOTAP, an enhanced WiFi access point that includes a downlink packet scheduling algorithm. Chapter 6 presents EAPS, to adjust the sleep duration of stations while they are expecting downlink packets. Chapter 7 presents SATRAC, a framework for traffic characterization of associated STAs on the AP for accurate TWT allocations Finally, Chapter 8 concludes this research and provides directions for future work.

This page is intentionally left blank

Chapter 2

Literature Review

2.1 Introduction

This chapter reviews the existing methods applicable to the objective of this thesis. Therefore, this chapter has three main sections organized as summarized in Figure 2.1:.

2.2 Power Saving Optimizations in WiFi Networks

2.2.1 Power Saving Methods in 802.11 (a/b/g/n/ac) Standards

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 [43, 44]. To reduce communication delay with AP, APSM requires a station waiting for downlink packets to

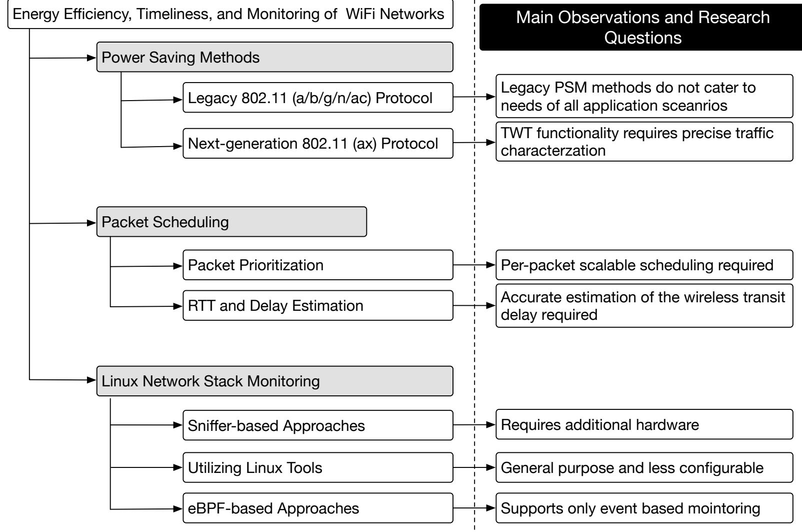

Fig. 2.1: The structure and main observations of the literature review conducted with respect to the contributions in this thesis.

stay awake for a *tail time* duration (e.g., 10 ms [7]) after each packet exchange [16]. 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 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 [45]. A new power-saving mechanism introduced in 802.11ax is 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) [46, 47]. 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.

2.2.2 Power Saving Methods 802.11 (ax) Standard

In addition to the power saving methods available in 802.11 (a/b/g/n/ac), TWT functionality introduced in 802.11ax protocol allows the STAs and AP to negotiate their sleep-wake schedule. This enables the STAs to wake up only during the scheduled TWT sessions. The two broad categories of TWT sessions are *individual* and *broadcast*, where the former is an agreement between the AP and a station, and the latter is an agreement between the AP and multiple stations. A TWT agreement could be used for UL, DL, or both types of communication. The two most important parameters determined during the negotiation process are: (i) *wake time*: the next time the station must wake up for the first TWT session, (ii) *wake interval*: the time interval between successive wake-ups, and (iii) *wake duration*: the amount of time the station stays awake every wake interval. The minimum value allowed is 256 μs. If the parameters are only valid for a single TWT session, the agreement is referred to as *explicit TWT*. In contrast, an *implicit TWT* session uses the defined parameters to repeat the sessions periodically (unless a new set of parameters are announced). Inside a TWT session, the AP can either send a trigger frame and assign sub-carriers to the stations, or the session can be non-triggered enabled. Also, inside a TWT session, the AP may leverage the *grouping* mechanism to assign time slots to each station.

TWT reservations are station-initiated, where each station simply requests TWT schedules based on their respective traffic patterns. This method either requires support from the applications or the driver modifications are required to provide application-agnostic traffic pattern recognition. Thus, station-initiated includes additional overhead on the computing resources or modification to the driver. Additionally, each station would in non-coordinated fashion However, the second method of allocating TWT sessions is AP initiated TWT, where the AP is in charge of assigning TWT sessions. In AP-initiated TWT, the AP being a centralized entity, has a global over

view of the entire network, the AP can allocate TWT schedule that works best for all. We have verified that two of the most popular 802.11ax cards (Intel®AX-200 and Qualcomm®Atheros QCA6391) comply with AP-initiated TWT.

Yang et al. [37] proposed two methods: the first one strives to maximize the throughput of all stations, and second one establishes proportional fairness. Using simulation, they compared the performance of these methods against the basic 802.11 channel access method and showed that the max-throughput method results in highest throughput. Their proposed solution allocates each Service Period (SP) to only one station. Qiu et al. [48] proposed a cross-layer TWT parameter selection methods dependent on the category of the flow and its latency requirements. Utilizing the traffic characteristics, such as inter-packet arrival time, they first categorize application type using machine learning methods. Based on the amount of TWT service period utilized by the STAs, they adaptively reconfigure the the TWT parameters on the fly. Since the proposed method is complicated and process-intensive, it has been implemented on high-end smart phones instead of IoT devices. Nurchis and Bellalta [33] provide an overview of TWT and its various features. They evaluate the number of messages needed for establishing periodic and aperiodic individual and broadcast TWT sessions. They show that DCF results in lower latency when the traffic rate is less than 4 Mbps per station. For higher traffic rates, using TWT allows the AP to schedule concurrent uplink transmissions, while the STA can perform better packet aggregation because of packet accumulation between TWT sessions. Chen et al. [34] address the problem of scheduling uplink TWT service periods to achieve both energy efficiency and high throughput. Power saving and packet scheduling methods for 802.11ac and earlier standards have been proposed in several works [18, 49, 50]. Our work is orthogonal to these methods and can be leveraged to further enhance the efficiency of TWT scheduling.

2.3 Packet Scheduling

2.3.1 Traffic Prioritization

Since 802.11 networks are used for the exchange of both elastic and real-time data, the 802.11e standard provides various access categories (AC) [51]. The AC of MAC frames is determined based on the differentiated services code point (DSCP) field[1] of IP header. This field comprises of different flagged bits, which when set, conveys to the lower layers the flow type and enforces IP precedence for per-hop QoS and priority. This layer-3 field is then mapped to the class of service (CoS) field in the MAC header [52]. By using CoS mapping, the kernel sets the priority socket buffer and enqueues the packet to the corresponding transmit queue. In the 802.11e protocol, each EDCA queue behaves as a virtual station and contends for the channel independently according to the contention parameters specified by the 802.11e standard.

Additionally, almost all Linux systems are capable of administering the manner in which the packets are sent over a medium by employing queueing disciplines known as *qdisc*. Situated between layer-2 and layer-3 of IP stack, qdisc provides several types of traffic scheduling (what packet to be sent) and traffic shaping (how many packets to be sent per time unit) mechanisms to hand packets over to layer-2.

Packet scheduling mechanisms such as [53] and [54] propose solutions to group stations such that only a few stations wake up to receive the packets after each beacon interval. Similarly, [55], [56], and [57] group the stations to limit the level of channel access contention by stations at any given point in time.

Kwon and Cho [58] assume that the AP is aware of the priorities of the registered stations. They prioritize packet reception according to the priorities of the stations

[1] This field is also know as the type of service (ToS).

based on their profile information such as the remaining power level of each station. To reduce the waiting time (and energy consumption) of PSM stations, Rozner et al. [31] enhance the access point to transmit the packets of these stations before those stations that do not utilize PSM. The downlink packet scheduler proposed in [59] prioritizes burst packet delivery after every beacon interval when a station wakes up to receive the buffered packets at the AP based on historical data and attention fairness. Clients with smaller attention requests are serviced before the others, thus allowing them to spend less energy to get one unit of attention, while allowing the clients with larger attention requests to sleep more compared to other popular scheduling methods, such as priority round robin and priority first come first serve.

Pyles et al. [28] utilize an application classifier to increase the priority of interactive Android applications. When a high priority application is detected, the WiFi driver uses the Adaptive PSM mechanism if the rate of packet exchange is beyond a certain threshold.

Wamser et al. [60] address resource allocation to multimedia traffic on home gateways. If the video or audio local playback buffer falls below 25s, its flow is moved to a higher priority queue, and once the buffer reaches 40s, the flow is downgraded to a lower priority queue. Flaithearta et al. [61] proposed an intelligent AP for VoIP traffic to address intra-AC prioritization among VoIP flows. This method finds the VoIP quality using the ITU-T E-Model [62] and the AP sets the DSCP value of the packets based on network characteristics collected from the RTP Control Protocol (RTCP). By using the additional higher priority AC_VO transmission queue provided by the 802.11aa standard, they divert a few VoIP calls to prioritize them over others. Qazi et al. [63] propose fine-grained mobile application detection using a machine learning trainer based on a decision tree ML tool in the control plane. To this end, they use `netstat` logs from employee devices along with the flow features (first N packet sizes,

port numbers, IP address range). In [64], the router dynamically adjusts bandwidth allocation of flows using Linux's `tc` utility. The aggregated bandwidths are computed for video, web browsing, file transfer, and voice classes using linear utility function [65] on account of the contextual-priority reports sent by the clients. The authors in [66] have demonstrated application-aware networking for video streaming. They identify characteristics of the flows through deep packet inspection and forward them via least congested links by dynamically changing the routing paths. They have compared the performance of bandwidth-based and DPI-based path selection mechanisms regarding *buffered playtime* in an SDN-enabled network. Afzal et al. [67] proposed a context-aware resource allocation scheme in wireless multimedia sensor-based WLANS. This method formulates an optimization problem on the basis of the service requirements of each flow, and allocates appropriate bandwidth and TXOP to the stations.

2.3.2 RTT and Delay Estimation

Peck et al. [68] 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 Delay (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 [68] 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. [16] 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 [68] nor [16] 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 [16] would not be effective in IoT scenarios where most of the bursts are short-lived. Sui et al. [69] 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. [70] 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 [18] 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. [31] propose NAPman, which prioritizes the delivery of PSM traffic as long as other stations are not affected. Tozlu et al. [4] show that increasing AP load has a higher impact on packet loss and RTT, compared to out-of-band interference. Pei et al. [14] 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. [49] 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.

2.4 Linux Network Stack Monitoring

Collecting monitoring data (a.k.a., network inspection, statistics collection) from WiFi APs makes it possible to study network operation [12, 13, 14, 15], enhance performance [71, 72, 73, 74], and secure these networks [19, 20, 21, 22]. Measurements reflecting the state of the network enables making informed decisions on APs, central controller, and stations. However, the existing works primarily rely on collecting monitoring data at the application and transport-layer levels. Additionally, they rely on lower sampling rate of collecting monitoring data; whereas, for applications such as delay prediction and packet scheduling, efficient and high-rate collection of monitoring data is necessary.

2.4.1 Sniffer-based Approaches

Using sniffer to capture network traffic has been widely used for network monitoring [75, 76, 77, 78, 79, 16]. Manufacturers have adopted this method in commercial deployments as well. For example, Cisco Meraki's MR18 APs include integrated sniffers. However, sniffer-based approaches add additional cost as they require dedicated hardware and software resources. Specifically, once an additional WL-NIC is added to an AP, each incoming packet requires the operating system to process an additional packet. Furthermore, the monitored data obtained from the sniffer provides only higher-level information about the traffic patterns and network characteristics. For example, this approach does not provide any insight into the delays incurred at each stage of packet processing. Also, depending on the processing resources available to the sniffer, it may miss capturing and logging some packets in dense environments and thus, multiple sniffers might be required capture all the packets [80]. In particular, any difference between the location, type, and antenna gain of the sniffer and those of the AP's WL-NIC result

in discrepancies in the way the two WL-NIC perceive the channel.

2.4.2 eBPF (enhanced Berkeley Packet Filter)

The importance of eBPF has been highlighted by both academia and industry in various application domains such as system monitoring, enhancing security, and supporting programmability. Contrary to modifying or adding kernel modules, eBPF programs are verified before being loaded in the kernel, which makes it a safer mechanism for accessing kernel memory [81]. Hence, the networking industry is relying on eBPF in production environments for system profiling and enhancing network functions by adding programmability to the data plane. For example, Facebook's *Katran* [82] and Sysdig's *Falco* [83] use eBPF for implementing a layer-4 load balancer and a Kubernetes runtime security and monitoring tool, respectively. Netflix has developed *Flow Exporter* for observing the per-flow transport layer (i.e., TCP/IP) statistics, and they reported that the overhead of using the Flow Exporter tool is less than 1% of the processor time; thereby enabling scalability over many Amazon Web Services (AWS) instances. One of the most notable technologies enabled by eBPF is eXpress Data Path (XDP) [84], which allows to tap into the reception path of the network stack prior to any memory allocation, resulting in significant performance benefits compared to other packet processing mechanisms. XDP is actively being used in various domains such as Distributed Denial-of-Service (DDoS) mitigation and packet inspection at Cloudflare and Facebook. Despite the wide adoption of eBPF, our work is the first to utilize eBPF for monitoring WiFi networks.

2.4.3 Utilizing Linux Tools

Most WiFi NIC drivers are implemented either as an extension of the kernel or as a loadable kernel module; thus, encapsulating hardware resources within the protected kernel space memory. To access the data maintained by the NIC and driver, user-space programs rely on network management tools such as `iwconfig`, `iwpriv`, `ethtool`, and `ifconfig`. These tools generally use `ioctl`, `sysfs`, or netlink for the communication between user-space processes and kernel.

`ethtool` allows for monitoring and configuring NIC. This tool constitutes a user-space module and a kernel-space module. These two modules communicate via `ioctl`, which extends the native Linux system call operations by providing functions for hardware configurations that use predefined data structures. These functions can be modified when new functionalities are added to the hardware. However, this may result in non-backward-compatible updated functions. Thereby, `ioctl` is not user friendly and it is difficult to extend. The information provided by `ethtool` with the `-S` (-`statistics`) option can be helpful in obtaining channel state information along with driver statistics such as the number of packets transmitted or received by the NIC. For example, Airtime Utilization (AU) can be retrieved using `ch_time_busy` and `ch_time`. However, the data retrieved by `ethtool` is not extensive and does not include some of the essential data about network performance. For example, per-station statistics, state of queuing discipline (qdisc), and NIC's register values, such as current NAV or failed FCS count, cannot be collected. Additionally, `ethtool` retrieves all the counters at once, preventing the user from specifying the list of measurement parameters. This results in the extra overhead of polling additional registers and driver's data structures, as well as searching for the required information in the collected data by user applications. Additionally, although the NIC updates AU per its internal clock cycle, `ethtool` reports AU time in milliseconds granularity, which results in lower accuracy and higher reporting

delay.

SoftMAC-based wireless drivers commonly provide a debug mode utilizing Virtual File System (VFS). Unlike regular files that reside on disk, VFSs (e.g., `sysfs`, `procfs`, and `debugfs`) reside in the main memory and facilitate communication between user-space and kernel-space. These interfaces remain empty unless a user-space process requests the resources. When requested, the kernel gathers the required measurements and populates the interface. For example, the ath9k driver utilizes `regdump-debugfs` to retrieve the values stored in all registers exposed to the driver by the NIC. A major shortcoming of `debugfs` is that it does not allow to query an arbitrary set of registers of the NIC. This results in a large number of unnecessary PCIe bus interrupts. Furthermore, data transfer size is limited to one memory page [85].

This page is intentionally left blank

Chapter 3

MonFi: A Tool for High-Rate, Efficient, and Programmable Monitoring of WiFi Devices

3.1 Introduction

The increasing number of WiFi devices, their stringent communication requirements, and the need for higher energy-efficiency mandate the adoption of novel methods that rely on monitoring the WiFi communication stack to analyze, enhance communication efficiency, and secure these networks.

In this chapter, we present MonFi, a Linux-based, open-source tool for efficient, high-rate, and programmable monitoring of WiFi devices[1]. This tool allows for monitoring the complete WiFi stack—NIC, driver, `mac80211`, `cfg80211`, `hostapd`, and qdisc. The monitoring frequency and the type of measurements collected can be programmatically specified using the user-space component of MonFi. We present methods to monitor the WiFi stack, and also implement and study methods for reducing the overhead of kernel to user-space communication and stabilizing monitoring rate in the presence of interfering loads on the processor. Through empirical evaluations, we show the higher efficiency of MonFi in terms of monitoring rate, stability of monitoring rate, and processor utilization, compared to the existing tools. For example, MonFi achieves about 28%

[1] The implementation of MonFi can be found at the following link: https://github.com/SIOTLAB/MonFi

higher monitoring rate and 16% lower processor utilization compared to `debugfs`, which is a widely-used method for monitoring communication stack. Compared to `ioctl`, the monitoring rate of MonFi is 21x faster.

As a publicly-available open-source tool, MonFi offers new opportunities in developing WiFi systems, from smart homes to enterprise networks to real-time industrial systems. Besides, MonFi reduces the development costs associated with using additional measurement devices. For example, by providing per-station information, MonFi eliminates the need for using additional hardware tools to analyze stations' power status.

The rest of this chapter is organized as follows: Section 3.2 overviews the literature and the tools used to monitor the WiFi stack. We present the design and implementation of MonFi in Section 3.3. Section 3.4 presents performance evaluation results. We summarize the chapter in Section 3.5.

3.2 Existing Monitoring Tools

Most WiFi NIC drivers are implemented either as an extension of the kernel or as a loadable kernel module; thus, encapsulating hardware resources within the protected kernel space memory. To access the data maintained by the NIC and driver, user-space programs rely on network management tools such as `iwconfig`, `iwpriv`, `ethtool`, and `ifconfig`. These tools generally use `ioctl`, `sysfs`, or netlink for the communication between user-space processes and kernel.

`ethtool` allows for monitoring and configuring NIC. This tool constitutes a user-space module and a kernel-space module. These two modules communicate via `ioctl`, which extends the native Linux system call operations by providing functions for hardware configurations that use predefined data structures. These functions can be

modified when new functionalities are added to the hardware. However, this may result in non-backward-compatible updated functions. Thereby, `ioctl` is not user friendly and it is difficult to extend. The information provided by `ethtool` with the `-S` (-`statistics`) option can be helpful in obtaining channel state information along with driver statistics such as the number of packets transmitted or received by the NIC. For example, AU can be retrieved using `ch_time_busy` and `ch_time`. However, the data retrieved by `ethtool` is not extensive and does not include some of the essential data about network performance. For example, per-station statistics, state of qdisc, and NIC's register values, such as current NAV or failed FCS count, cannot be collected. Additionally, `ethtool` retrieves all the counters at once, preventing the user from specifying the list of measurement parameters. This results in the extra overhead of polling additional registers and driver's data structures, as well as searching for the required information in the collected data by user applications. Additionally, although the NIC updates AU per its internal clock cycle, `ethtool` reports AU time in milliseconds granularity, which results in lower accuracy and higher reporting delay.

SoftMAC-based wireless drivers commonly provide a debug mode utilizing VFS. Unlike regular files that reside on disk, VFSs (e.g., `sysfs`, `procfs`, and `debugfs`) reside in the main memory and facilitate communication between user-space and kernel-space. These interfaces remain empty unless a user-space process requests the resources. When requested, the kernel gathers the required measurements and populates the interface. For example, the ath9k driver utilizes `regdump-debugfs` to retrieve the values stored in all registers exposed to the driver by the NIC. A major shortcoming of `debugfs` is that it does not allow to query an arbitrary set of registers of the NIC. This results in a large number of unnecessary PCIe bus interrupts. Furthermore, data transfer size is limited to one memory page [85].

Using netlink addresses some of the aforementioned challenges. Firstly, since

it is a socket-based mechanism, netlink can be initiated by both kernel-space and user-space processes, whereas, VFSs and `ioctl` can be instantiated only by user-space processes. This is particularly useful for event-based data collection. For example, consider a scenario where the kernel sends measurements to a user-space application whenever a packet is received. Secondly, netlink facilitates asynchronous communication by storing messages in queues and initiating the receiver's reception handler. The receiver can process the information synchronously or asynchronously. In contrast, `ioctl` and VFSs communicate synchronously. netlink can also multicast to multiple processes at once, whereas, `ioctl` and VFSs support unicast messages only.

3.3 Design and Implementation of MonFi

In this section, we identify the measurements that are indicative of network operation and dynamics, and propose methods for collecting these measurements from various components of the networking stack. We also present methods to achieve monitoring rate determinism. Finally, we discuss the monitoring modes provided by MonFi.

3.3.1 Architecture

Figure 3.1 presents the architecture of MonFi. The Controller is a user-space module that configures and receives measurements from the Collector. The Collector is a kernel-space module that collects data across the network stack. The Collector is implemented as a part of the driver to share a set of functions and data structures. The Collector also interacts with other modules of the communication stack.

In general, current 802.11 drivers are categorized as either SoftMAC or FullMAC. SoftMAC drivers implement a part of the MLME in software, thereby utilizing

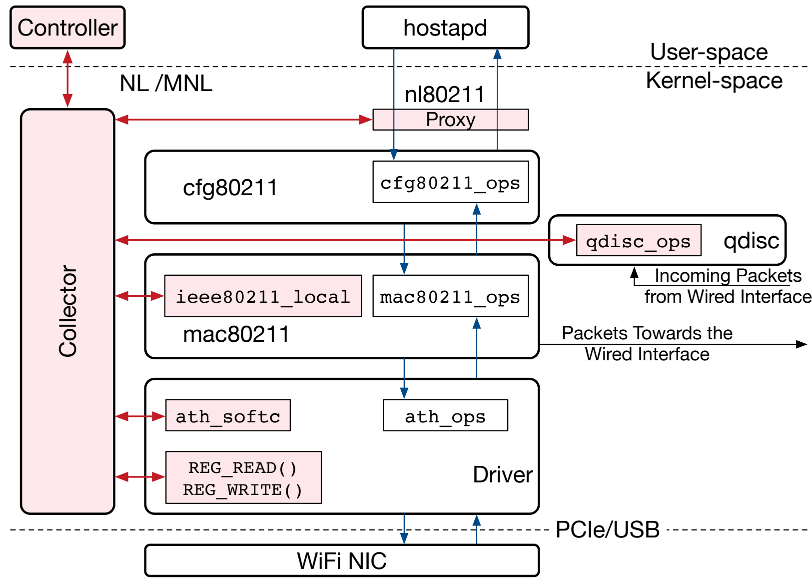

Fig. 3.1: The architecture of MonFi. The Controller allows use-space programs to specify the type, rate, and method of collecting measurements across the WiFi stack. The Controller collects the requested data from the stack. Various kernel modules have been modified to facilitate collecting monitoring data from them. Note that the `hostapd` daemon is only used when MonFi runs in an AP.

the host system's processing resources. Only time-critical MAC functions, such as managing timeouts, inter-frame spacing, and channel access backoff, are implemented in the hardware. Whereas, FullMAC utilizes an additional processor on the NIC to implement all MAC functions within the firmware. Currently, most commercial WiFi adapters rely on a SoftMAC architecture [38], especially considering the ease of updating. Thus, in this chapter, we assume the underlying WiFi NIC's driver is a SoftMAC.

To perform the standard AP functionalities, we use `hostapd` [86], which is used by Commercial Off-The-Shelf (COTS) APs. `hostapd` is a user-space daemon that handles beacon transmission, authentication, and association of stations.

3.3.2 Extensive Data Collection Across the Protocol Stack

In this section, we identify and discuss the process of monitoring NIC, driver, qdisc, and the other components of the WiFi stack.

3.3.2.1 PHY and MAC measurements

WiFi NICs provide I/O memory regions consisting of registers that can be accessed via the PCIe bus. These registers allow to configure the NIC and obtain its internal status. Accessor functions, such as `ioread32()` and `iowrite32()`, are utilized to *get* and *set* the status of the registers, respectively. The Collector utilizes these functions to extract register values from the NIC. For example, to compute AU, which reflects the level of activity on the wireless channel, the Collector fetches the values of `AR_CCCNT` and `AR_RCCNT` registers (available in Atheros chipsets). The former register stores the time elapsed since the start time of the NIC, and the latter stores the amount of time the NIC has been sensing activity on the channel. We denote these measurements as T and B, respectively. The AU during an interval $t1$ to $t2$ is computed as $(B_{t2} - B_{t1})/(T_{t2} - T_{t1})$.

Most COTS NICs (e.g., Netely NET-900M and Compex WLE900VX) operate with the reference clock speed of 44 MHz when using the 2.4 GHz band and 40 MHz when using the 5 GHz band (considering a 20 MHz channel), and update the values of the registers corresponding to the clock rate. The Collector retrieves the values of these registers, and then, either decodes them to decimal values and in milliseconds format, or stores the raw measurements that can be decoded asynchronously when needed. The Controller also provides the functionality that allows user-space applications to specify the list of registers or register addresses.

Compared to the legacy 802.11 standards (i.e., a/b/g), the recent standards

(i.e., n/ac/ax) offer numerous physical layer enhancements that result in bit rates beyond 600 Mbps. Some of these enhancements are concurrent Multiple-Input Multiple-Output (MIMO) streams, wider channel bandwidth, and higher-order Modulation Coding Schemes (MCSs). These parameters can be configured using `hostapd`'s configuration file or `hostapd_cli reconfigure`. `hostapd` utilizes netlink to communicate with the WiFi Configuration API—`cfg80211`. The `cfg80211` module acts as a bridge between user-space and kernel, and provides a unified interface in the form of callback functions to control the NIC. Each callback implemented in `cfg80211` is associated with a corresponding function in the driver to configure the NIC. The Collector intercepts the netlink events triggered by `hostapd` to keep track of any modifications applied to the NIC.

The physical layer configuration of the NIC may not necessarily represent the parameters used by each AP-station pair when communicating. For example, even when an AP announces supporting 40 MHz channels, the station may not support this channel width, and instead use a 20 MHz channel. Also, the two ends of a communication link dynamically change their MCS, based on multiple factors such as Received Signal Strength Indicator (RSSI) and retransmission rate. Therefore, each link's physical layer parameters are essential for characterizing communication efficiency in terms of metrics such as throughput and Packet Error Rate (PER) on a per-station basis. To this end, MonFi collects RSSI, MCS, and the number of MAC layer retransmissions, *on a per-station basis*, as follows. The kernel maintains a global list of devices utilizing the `net_device` structures in `mac80211`. The `*priv` field contains driver-specific structures and maintains the statistics for respective stations. This field is represented by `ath_softc` in Atheros NICs. The Controller module allows user-space applications to specify the stations that must be monitored. The Collector collects per-station measurements requested by the Controller by polling the corresponding fields of the `softc`

data structure and reports them to the Controller along with the MAC address of the station. In order to avoid data copy, the pointer to the `softc` data structure is shared between all tasks in the driver. However, race condition happens when multiple tasks in the driver try to access `softc` concurrently. We utilize a mutex lock before accessing `softc` to avoid race conditions.

The 802.11 standard supports various power-saving methods to allow stations to switch to sleep mode and conserve energy. The power state of each associated station is maintained by `mac80211` (cf. Figure 3.1). Whenever a PS-POLL or Null packet is received by the AP, `mac80211` notifies the respective driver about the change in power state via the `drv_sta_notify()` function. Whenever the driver receives an update, the Collector forwards the updated power state and the MAC address of the station to the Controller.

To reduce the overheads pertaining to channel access and PHY and MAC headers, the MAC layer of high-throughput WiFi standards (i.e., 802.11n/ac/ax) aggregates multiple MAC protocol data units (MPDUs) into an Aggregated-MAC protocol data unit (A-MPDU). MonFi monitors the number of packets aggregated in each A-MPDU, the number of MPDUs currently enqueued in each queue of the driver, and the instance each packet is sent, as follows. Frame aggregation is performed in the software queues maintained in the driver. These queues act as buffers between the NIC's queue and the upper layers in the protocol stack. When there are multiple packets in a queue of the driver, they are aggregated into a single MPDU. NIC notifies the driver via a callback function after processing each packet from the hardware queue. For example, in Atheros NICs, `ath_tx_tasklet()` is called for informing the driver to process the next packet for transmission.

3.3.2.2 Monitoring qdisc

In Linux systems, there is a qdisc assigned to each NIC to buffer egress traffic. The Linux kernel introduces a rich set of queuing disciplines between the network subsystem in the kernel and `mac80211`, enabling a flexible traffic control framework. The efficiency of the qdisc layer is dependent on its packet scheduling method, the size of the queues, the rate of incoming traffic, and the rate of WiFi transmission. When the queue size is small, the qdisc layer may not be able to absorb bursts of incoming traffic while waiting for wireless transmission, thereby causing packet drops. Alternatively, longer queues may cause long end-to-end delays (a.k.a., bufferbloat [12]). Given the high impact of qdisc on packet scheduling and delay, we collect the number of packets currently enqueued in each queue of the qdisc layer. By default, every network interface is assigned a pfifo_fast qdisc as its transmission qdisc. This mechanism contains three bands, and dequeuing from a band occurs when the upper bands are empty. Each variant of qdisc is implemented as a kernel module (in `/net/sched`) and contains `.enqueue` and `.dequeue` functions. In MonFi, we modified these kernel modules to report the status of qdisc by logging the number of packets currently enqueued in each band. Our current implementation supports pfifo_fast and PRIO qdiscs. This method can easily be applied to other qdiscs such as Hierarchical Token Bucket (HTB).

3.3.3 Monitoring the Host system

In an ideal use-case, the behaviour of packet reception and transmission can be determined with the help of the data collection methods described earlier in this section. However, we need to note that the tasks performed by various components of the networking sub-systems (e.g., `mac80211`, driver) and the additional processes introduced by MonFi are scheduled by the operating system to run on the processor cores available

in the system. Hence, we evaluate the available and occupied computational resources. Depending on the available hardware resources, this allows the user to specify monitoring parameters (e.g., monitoring rate) that do not impose high processing overhead. Also, MonFi allows to keep track of the latency between requesting and receiving a measurement by the Controller. For real-time systems that react to network dynamics, this latency determines the validity and usefulness of the measurements obtained.

3.3.3.1 Dedicating computing resources to MonFi

Most modern processing platforms are based on Symmetric Multi-Processing (SMP) architecture consisting of multiple physical processor cores that are capable of executing multiple threads concurrently. User-space threads, software interrupts, and hardware interrupts are evenly scheduled to be served by processor cores. Hence, the performance of MonFi can be easily interfered by background processes and excessive context switching.

To address this problem, we bind the execution of MonFi's components with an isolated processor core, such that no other processes or interrupts are scheduled on this core. For example, consider an Intel i5 processor that includes two physical cores, PC1 and PC2, where each core contains two logical cores, LC1/LC3, and LC2/LC4, respectively. Utilizing `isolcpus=1,3` kernel boot parameter, we isolate the physical core PC1 for operating system and other background tasks. Thus, the physical core PC2 is dedicated to the execution of MonFi's components. To this end, first, LC2 is dedicated to the Controller via `sched_affinity()` or `taskset` system calls. Second, LC4 is dedicated to the driver. Since the Collector is an extension of the driver, it runs on LC4. Since all the software interrupts executed as the result of hardware interrupts are scheduled on the same core [87, 88], all the software interrupts generated by the driver are also scheduled on LC4. Third, utilizing the `/proc/interrupts` file, we obtain all

possible components that can generate hardware interrupts and set the `smp_affinity` of these components to LC1 and LC3; thereby, no hardware interrupt will be scheduled on LC2 and LC4.

3.3.4 Sharing the Collected Data with User-space

The Collector shares its data with user-space for further processing. As discussed in Section 5.2, both `procfs` and `ioctl` are less efficient compared to netlink. Therefore, we use netlink sockets for communication between kernel-space and user-space. We simply refer to this method of Collector and Controller communication as MonFi w/ netlink (MonFi-NL). To further reduce the overhead of this communication, a memory-mapped region is established for netlink's receive and transmit buffers. These buffers are shared by the Controller and Collector to prevent data copying overhead. We refer to this mechanism as MonFi w/ mmaped-netlink (MonFi-MNL).

User-space applications can use the Controller to specify three types of monitoring modes: (i) Event-based Data Collection (EDC), (ii) Polling-based Data Collection (PDC), and (iii) Event and Polling-based Data Collection (EPDC), which is a hybrid of EDC and PDC. We explain these modes as follows.

3.3.4.1 Event-based data collection (EDC)

In this mode, the Collector sends monitoring data to the Controller whenever an event occurs. The event type is specified by the Controller. For example, sample events causing the NIC to generate an interrupt are packet reception, channel availability after Distributed Inter Frame Space (DIFS), and the expiration of software beacon alert timer (a.k.a., `bcntimer` used to trigger periodic beacon transmission). NIC interrupts are handled by the driver to determine the interrupt type and call an appropriate tasklet.

For example, with Atheros NICs, `ath9k_tasklet()` and `ath_isr()` handle interrupts and then call tasklets such as `ath_tx_tasklet()` and `ath_rx_tasklet()`. We have modified the driver to monitor these interrupts and trigger data collection whenever a match in the list of events provided by the Controller is found.

3.3.4.2 Polling-based data collection (PDC)

This mode is particularly useful in applications such as channel allocation, station handoff, packet scheduling, and intrusion detection. In this mode, the Controller collects data from the Collector in fixed or variable intervals, depending on the user-space application's demand. Variable intervals can be specified by using various distributions such as the Poisson distribution. We eliminate the overhead of user-space to kernel-space communication as follows. Measurement collection parameters are passed by the Controller to the Collector once. These parameters primarily include: (i) the Inter-Measurement Interval (IMI), either as a fixed value or the parameters of the distribution used to determine IMI, and (ii) the total number of measurements. Once received, the Collector generates reports according to IMI configuration. Also, we reduce the overhead of Collector to Controller by allowing the Controller to specify when data must be sent up. Specifically, instead of sending all measurements, the Controller can instruct the Collector to send upward reports only when the difference from the previously reported value is higher than a particular value. This threshold is specified on a per-measurement-type basis.

3.3.4.3 Event and Polling-based Data Collection (EPDC)

In addition to the set of measurements collected at the time of an event, this mode allows for collecting the past n measurements preceding the occurrence of the

event. We have implemented a circular queue in the Collector module. At each IMI, the Collector gathers and places a new measurement into the queue. Once an event occurs, the Collector sends the most recent measurement, as well as the values in the circular queue, to the Controller. This approach is particularly useful for time series analysis and neural network algorithms such as Long Short-Term Memory (LSTM), which require recent history of n measurements in the temporal domain to estimate the *trend* in a time series.

3.4 Performance Evaluation

In this section, we evaluate the monitoring rate and processing overhead of MonFi when used to monitor the operation of an AP. The wireless NIC used is Compex WLE900VX, which includes the QCA9880 chipset and supports 3x3 MIMO 802.11ac. The driver used is ath10k. The processor is a dual-core Core i5. A similar hardware configuration is used by the station associated with the AP.

3.4.0.1 MonFI vs ioctl and debugfs

We first evaluate monitoring rate and its effect on processor utilization. Using `debugfs`, MonFi-NL, and MonFi-MNL, we collect the following parameters: AU, power state of the associated station, and five registers of the NIC. Using `ioctl`, we only collect AU. These experiments were conducted in the presence of regular CPU load, which is less than 10% and is caused by the normal functioning of the operating system tasks and AP functionality.

Figure 3.2(a) shows measurement collection rate and Figure 3.2(b) shows average CPU utilization as a function of IMI. As we reduce IMI, the monitoring rate

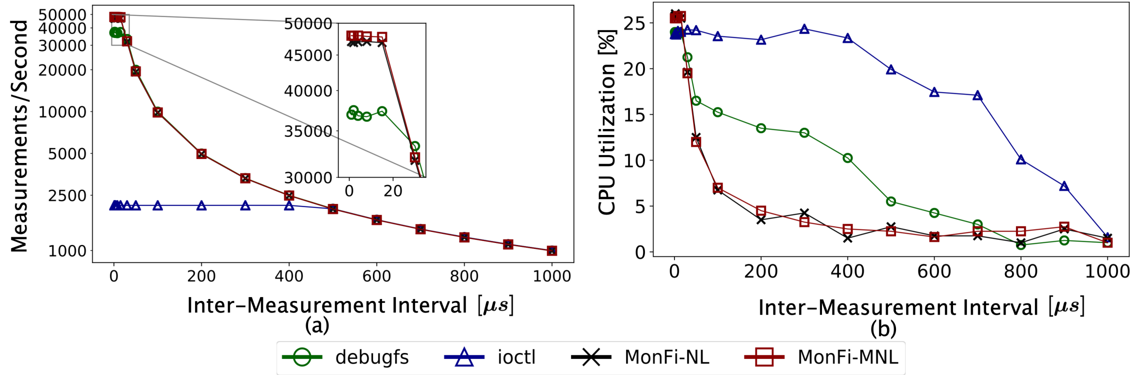

Fig. 3.2: (a) Measurement collection rate (per second) and (b) average CPU utilization as a function of IMI when using `debugfs`, `ioctl`, MonFi-NL, and MonFi-MNL. The monitoring rate of MonFi-MNL is 28% faster than `debugfs`.

achieved with each tool increases up to a certain point. The highest monitoring rate is provided by MonFi-MNL at 48005; whereas, `ioctl`, `debugfs`, and MonFi-NL plateau at 2115, 37481, and 47033 measurements per second, respectively. Although `ioctl` only collects AU, it demonstrates significantly lower monitoring rate and higher CPU utilization over all IMI values, compared to the other tools. Specifically, the monitoring rate of `ioctl` is 21x slower compared to MonFi-MNL. Compared to MonFi-MNL, the highest measurement collection rate provided by `debugfs` is 22% lower and its processor utilization is 20% higher, on average. Considering the higher performance of MonFi-NL and MonFi-MNL, we only consider these tools in the rest of studies presented in this section.

3.4.0.2 Impact of processor load on monitoring performance

We now evaluate how a higher processor utilization caused by a user-space daemon affects monitoring rate. This represents a scenario where a daemon is performing real-time data analysis and decision making. We utilize the `stress-ng` tool to generate a synthetic processor load. We evaluate using both dedicated and non-dedicated cores for the execution of MonFi processes, as explained in Section 3.3.3.1. Note that

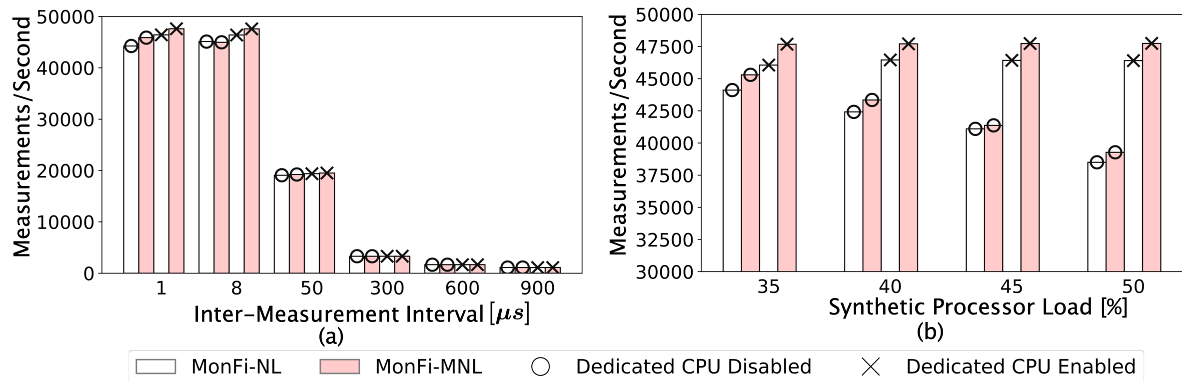

Fig. 3.3: (a) Measurement collection rate in the presence of a synthetic 30% processor load. (b) Maximum monitoring rate when the synthetic processor load increases from 35% to 50%. Based on these results, using dedicated cores to achieve deterministic monitoring performance is essential.

when using dedicated cores, the synthetic load is not scheduled on the cores assigned to MonFi. Figure 3.3 shows that reducing IMI and increasing processor load affect monitoring rate when the cores are shared. Considering MonFi-MNL, Figure 3.3(a) reveals that using dedicated cores increases the maximum monitoring rate by 1725 compared to shared cores. Figure 3.3(b) shows that using dedicated cores results in sampling rate stability versus processor load. When the synthetic load is increased from 35% to 50%, using dedicated cores achieves a constant monitoring rate of about 47000, while shared cores drops monitoring rate from 44707 to 38891, on average. These results confirm the importance of using MonFi with dedicated cores when a stable monitoring rate is required. Figure 3.3(b) also demonstrates the impact of using mmap-netlink instead of netlink to achieve a higher monitoring rate. For example, when the synthetic load is 35%, using MonFi-MNL collects 1621 and 1186 measurements higher than MonFi-NL with and without using dedicated cores, respectively.

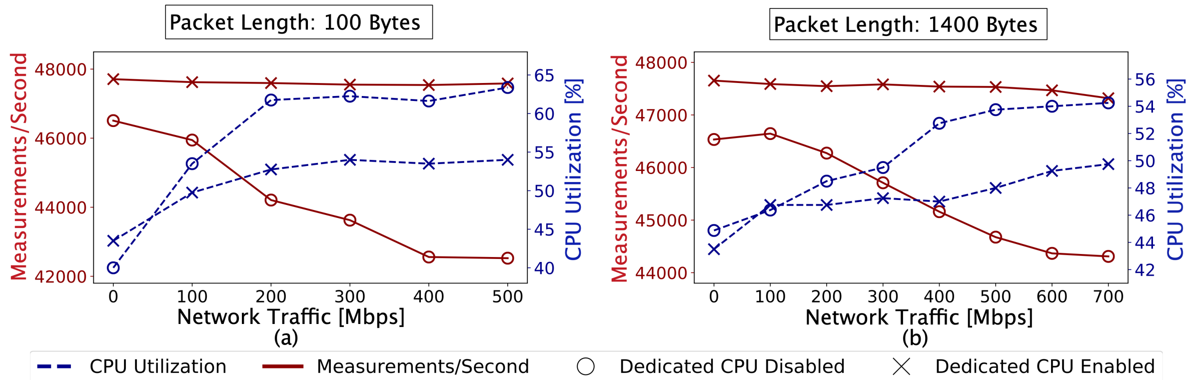

Fig. 3.4: Measurement collection rate (left y-axis) and CPU utilization (right y-axis) versus packet switching rate for 100-Byte (a) and 1400-Byte (b) packets. These results show that using dedicated cores is necessary to ensure deterministic monitoring rate in the presence of packet switching overhead.

3.4.0.3 Impact of packet switching on monitoring performance

In this section, we study the effect of packet switching on the monitoring rate of MonFi-MNL. Figures 3.4(a) and (b) show the results for 100-Bytes and 1400-Bytes packets being switched by the AP. These packets are received over the wireless interface (using the 802.11ac standard) and then switched to the wired interface. We were able to achieve the maximum throughput of 500 Mbps and 700 Mbps for 100-Byte and 1400-Byte packets, respectively. Reducing packet size results in a lower throughput due to the higher overhead of header transmission and waiting time caused by channel access back-off. For a given throughput rate, using smaller packets increases AP's processing overhead, which is caused by the higher number of interrupts, header processing, and packet copying operations. As both figures show, using shared cores results in a significant drop in the number of measurements per second. For example, when using 100-Byte packets, increasing the AP's switching rate from 100 Mbps to 500 Mbps causes the number of measurements per second to drop from 46508 to 42525. In contrast, using dedicated cores shows a relatively steady behaviour (less than 1% variation).

Our results confirm that MonFi can be used on average-grade dual-core APs for

very high speed, efficient WiFi stack monitoring. With the denser deployment of APs and the prevalence of the 802.11ac and ax standards, the need for microsecond-level monitoring escalates.

3.5 Summary

With the higher number and demand of WiFi devices, supporting efficient, high-rate monitoring of the WiFi stack is an essential requirement to analyze network operation, enhance performance, and enforce security methods. In this chapter, we presented MonFi, which allows user-space applications to specify the type and rate of collecting measurements. Our empirical performance evaluations confirm the higher sampling rate and processing efficiency of MonFi compared to the existing Linux tools.

This page is intentionally left blank

Chapter 4

FLIP: A Framework for Leveraging eBPF to Augment WiFi Access Points and Investigate Network Performance

4.1 Introduction

Despite its importance, understanding the operation of the WiFi stack in various settings is a challenging undertaking for the research community. The WiFi networking stack is complex and includes multiple layers across the WL-NIC, driver, Linux kernel modules, and user-space daemons. Although there exist tools that provide visibility into some of these layers, the performance and range of visibility of these tools are far from what is needed to analyze these networks and design solutions for performance enhancement effectively. Due to this shortcoming, a large number of existing works rely on simulation. Also, when high-rate monitoring is necessary, existing works rely on packet capture and static data analysis [19, 20, 21]. Another category of works relies on tools that have been primarily designed for *infrequent* monitoring and configuration [13, 14]. For example, Linux-based tools such as `iw` and `ethtool` can be used to collect *some* of the operational data from the WiFi stack; however, the sampling rate and efficiency of these tools are far below what is required for high-rate monitoring.

In this chapter, we present FLIP, a framework to augment the networking stack of Linux-based WiFi devices using the eBPF technology to collect a wide range of

monitoring data that can be used for both network operation investigation as well as developing methods that react to network dynamics. We first study the operation of the WiFi stack and then show how eBPF can be leveraged to interface with the components of the WiFi stack to monitor various aspects of network operation. We focus on two aspects of network performance analysis: *First*, since existing studies reveal the considerable effect of packet switching delay in APs [26, 89], we investigate and monitor the impact of queuing and channel access contention on the delay of switching packets from the AP's wired interface to the wireless interface. Using the FLIP framework, we build a testbed and study how various parameters such as traffic access category and the number of stations affect the switching delay. *Second*, we propose a novel method to track the duty-cycle pattern and energy consumption of stations. To this end, we rely on the fact that stations need to inform AP's whenever they change their power mode (sleep to awake and vice-versa), as mandated by the 802.11 power save mechanisms. Therefore, monitoring the driver's pertaining data structures allows for tracking stations' duty cycle pattern. This approach eliminates the need for external power measurement tools when studying the energy efficiency of resource-constrained stations. To show the effectiveness of this approach, we rely on empirical measurements and compare the accuracy of FLIP with a commercial power monitor. Our results show that the error of FLIP is 6% compared to a commercial power monitoring tool. We provide FLIP as a publicly available tool that can be implemented on off-the-shelf APs[1].

The rest of this chapter is organized as follows. We present the overall architecture of FLIP in Section 4.2. In Section 4.3, we first explain the approach employed to monitor packet switching delay from the wired interface to the wireless interface, and then present an empirical analysis of this delay. In Section 4.4 we show how FLIP can

[1]FLIP implementation can be found at the following link: https://github.com/SIOTLAB/FLIP

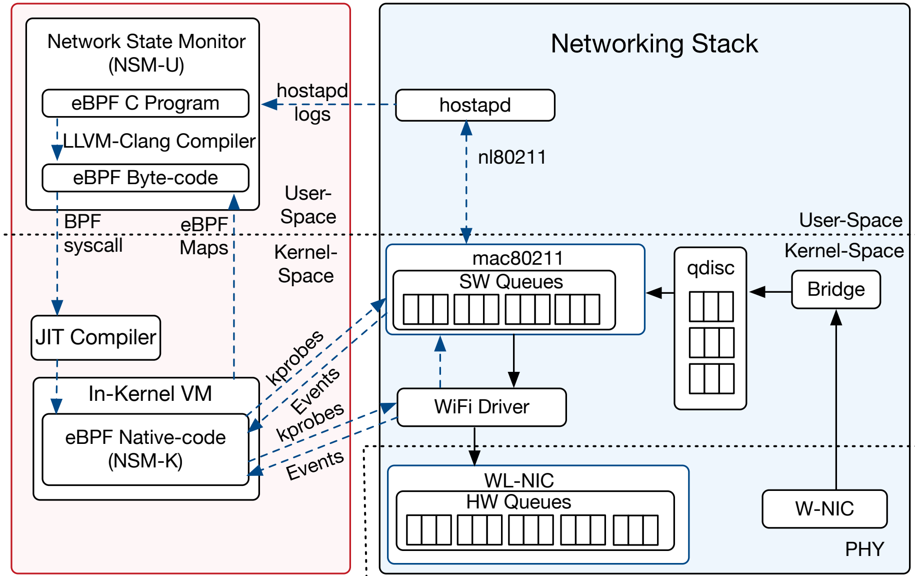

Fig. 4.1: The FLIP architecture for augmenting APs. The right side presents the network stack, and the left side shows the Network State Monitor (NSM) module that relies on eBPF to interact with the network stack. The dotted arrows denote the collection of monitoring data. The solid arrows represent the path taken by data packets (wired-to-wireless switching data-path).

perform passive energy monitoring of stations. We summarize the key contributions this work in Section 4.5.

4.2 System Architecture

The WiFi stack of commercial, off-the-shelf APs includes components that span WL-NIC, driver, Linux's kernel-space modules, and user-space daemons. In this section, we present the architecture of these APs and then explain how eBPF can be leveraged to enhance visibility into the WiFi stack.

4.2.1 AP's Networking Stack

Apart from switching packets between the Wired Network Interface Card (W-NIC) and WL-NIC, an AP is responsible for operations such as beacon generation and handling the association and dissociation of stations. These functionalities are enabled by several components including `hostapd`,`wpa_supplicant`, `mac80211`, driver, `qdisc`, and WL-NIC, as illustrated in the right-half of Figure 4.1. `hostapd` is a user-space daemon that handles authentication, association, and dissociation of stations. To generate control and management frames, `hostapd` configures `mac80211` and driver via the netlink (`nl80211`) library. The `mac80211` module provides a unified interface between the driver, `qdisc`, and `hostapd`. The `mac80211` module also administers the MLME functions for SoftMAC drivers; sample functions are building MAC headers and assigning sequence numbers. SoftMAC drivers implement a part of layer-2 functionalities in software utilizing host system's hardware and software resources. On the other hand, only time-critical MAC functions, such as managing timeouts, inter-frame spacing, and channel access backoff, are implemented in the WL-NIC. Currently, most commercial WL-NICs rely on the SoftMAC architecture [38], especially considering the ease of updating. Similarly, in this chapter, we assume the AP's driver is based on the SoftMAC architecture. Finally, the driver is responsible for packet aggregation and transferring packets to the WL-NIC for transmission on the channel.

4.2.2 Leveraging eBPF for Collecting Monitoring Data from the Kernel

The left-half of Figure 4.1 illustrates the extended Berkeley Packet Filter (eBPF) infrastructure. eBPF facilitates runtime patching of the kernel image by enabling the execution of user-defined logic when a system call or a kernel function is executed.

eBPF programs (written in C) are compiled into byte code utilizing the LLVM-clang compiler. This byte code is executed in an *in-kernel* virtual machine, and thus, reduces context-switching overhead. eBPF programs are attached to probe events (i.e., kprobe or a tracepoint) that mark as the instantiating points for the execution of user-defined logic. *Tracepoints* are required to be manually inserted into the kernel code by utilizing the `TRACE_EVENT()` macro. Whereas, *kprobes* are automatically defined in kernel's symbol table (`/proc/kallsyms`) along with their virtual addresses for almost all system calls and kernel functions that have been declared as neither inline or static. Hence, we utilize kprobes because it does not require any modifications to the kernel. For a particular function (or a system call) in the kernel, BPF system calls replace the instruction at the address of the function's execution with the breakpoint instruction (e.g., `int3` for x86 platforms). Whenever this breakpoint is hit, the context of the function is saved and the user-defined logic in the eBPF program is invoked. Once the execution of user-defined logic completes, kprobe executes the instruction that was replaced by the breakpoint and continues the kernel's normal execution path. eBPF also allows accessing the kernel-functions' arguments from user-space via eBPF data structures, a.k.a., eBPF Maps. eBPF Maps are transferred to the user-space via a ring buffer and can be accessed by high-level languages such as C, Python and Lua.

The Network State Monitor (NSM) is composed of two components: *NSM user-space (NSM-U)* and *NSM kernel-space (NSM-K)*. These components utilize eBPF to log the timestamps of the kernel functions that are invoked as packets traverse the stages of the networking stack. Furthermore, the NSM-K module also logs the state of the function arguments when the function is called for execution, and transfers it the monitored data to NSM-U via eBPF Maps. In the subsequent sections, we will elaborate on this method of obtaining monitoring data from the kernel.

4.3 Delay Analysis of Switching Packets from the Wired Interface to the Wireless Interface

Once a packet is received over the wired interface of an AP, the packet needs to pass through multiple queuing disciplines before contending for channel access. Specifically, the packet will need to contend with other flows both internally in the AP's queues and also physically during the CSMA process. Investigating packet switching delay from the W-NIC to WL-NIC is particularly difficult because it depends on various factors including the AP's traffic intensity, queuing disciplines used at various layers, airtime utilization by other APs and stations, and access category of flows. Recent studies show that the delay experienced at APs is more than 60% of the total communication delay between a station and server, and this delay is between 20ms to 250ms, depending on traffic congestion [14, 89, 90, 91]. The queuing disciplines employed by the Linux's qdisc, driver and NIC further complicate investigating and understanding the causes of this delay [12, 92, 90]. Vendors heuristically design device drivers' packet scheduling algorithms [12, 92], and the operation of non-open-source drivers is unknown.

In this section, we present a novel approach towards measuring and monitoring packet delay from the instance it arrive on the AP's W-NIC until it is transmitted successfully by the WL-NIC. We then utilize this framework for packet delivery delay analysis.

4.3.1 Power Saving Methods of 802.11

Since the amount of time spent by packets in the AP is affected by the power saving mode employed by the station, in this section we first overview the power saving mechanisms.

Two of the most widely adopted power save mechanisms are PSM and APSM. With PSM, each AP periodically (every 102.4 ms) sends a beacon packet, and stations wake up at beacon instances to check if the Traffic Indication Message (TIM) bit in a beacon is set. If the TIM bit is set, the station sends a PS-Poll packet to indicate its transition to awake state and to retrieve each of the queued packets from the AP. Once there are no more packet, station immediately transitions to sleep mode. The packets that arrive at the AP after the transition of the station into sleep mode are queued until the next beacon instance. For example, in a request-response scenario, whenever a station sends an uplink request to a server via the AP, the response received from the server will need to wait until the next beacon announcement. For delay-sensitive communication, the APSM method allows the station to remain in awake state for a fixed duration, known as *tail-time*, after each packet exchange with the AP. Considering the request-response scenario, if the response arrives before the tail time expiry, it is immediately delivered to the station.

4.3.2 FLIP's Methodology for Monitoring Wired-to-wireless Switching Path

The right-half of Figure 4.1 illustrates the modules along path taken between the W-NIC to WL-NIC. These delays are illustrated in Figure 4.2 and explained as follows.

4.3.2.1 Queuing Disciplines (qdisc)

Whenever a packet arrives on the AP's wired interface at time t_1, after the MAC address table entry lookup, the packet is transferred to the network stack. Residing between the bridge and the WiFi subsystem, qdisc implements a programmable set of

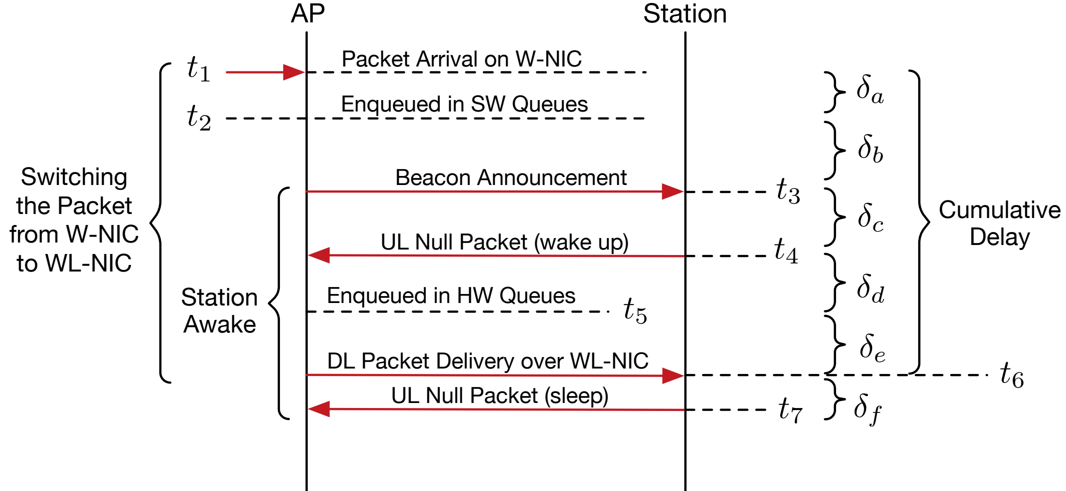

Fig. 4.2: Delay components of a data packet being switched from the wired interface to the wireless interface. The station uses the APSM energy-efficiency mode.

queues (a.k.a., bands), enabling a flexible traffic control framework. For example, every network interface is assigned a qdisc, which is pfifo_fast by default [90], consisting of three bands. The access category of each packet is inferred from the Type of Service (ToS) field in the IP header, then the packet is enqueued in one of these bands according to qdisc's *priority map*. For example, pfifo_fast qdisc's priority map specifies that a voice packet is enqueued in the first band, a video packet is enqueued in the second band, and background and best-effort packets are enqueued in the third band. A packet is dequeued from a band only when its higher-priority bands are empty. For example, unless the first band (corresponding to voice) is empty, packets in the second band (corresponding to video) are not dequeued. Hence, the queuing delay experienced by a packet enqueued in the lowest-priority queue depends on on the current utilization level of that queue as well as the utilization of higher-priority queues.

4.3.2.2 mac80211

Packets are dequeued from the qdisc queues and handed over to the WiFi networking subsystem, whenever lower layer queues are not full.[2] Specifically, if there is available space in the `mac80211` module, packets are dequeued from qdisc and inserted into mac80211 queues (at t_2), known as software (SW) Queues For each associated station, the SW Queues include a queue per access category. Each queue is resembled by a `ieee80211_txq` structure. The `mac80211` module performs the following functions whenever a packet is enqueued in its SW Queues at t_2. First, in case the destination station is in low-power sleep state, the `ieee80211_beacon_add_tim` function sets the TIM bit inside the next beacon to be sent at t_3. Second, the driver is notified of the pending packets via `drv_wake_tx_queue` function. Finally, `ieee80211_sta_register_airtime` structure updates the airtime fairness metrics maintained for each station. Recent works [12] show that qdisc can cause latencies of higher than 100 ms due to increased queue sizes (a.k.a., bufferbloat). To remedy this problem, they propose to disable qdisc, and instead, an integrated traffic control mechanism (commonly known as FQ_CODEL) based on *airtime-fairness* of associated stations has been proposed. Several WiFi drivers (e.g., ath9k, ath10k, rtl8723) use this approach and employ a round-robin dequeue scheduling mechanism for each intermediate SW Queue based on the FQ_CODEL algorithm. This ensures that each station is provided with a fair-share of the channel's airtime.

4.3.2.3 Driver and WL-NIC

The station conveys its transition into awake mode by sending a Null packet that its power-save-bit is '0'. At this point, the driver dequeues the packets from the SW Queues and passes them to the WL-NIC. WL-NIC includes five hardware (HW)

[2]qdiscs such as Token Bucket Filter (TBF) and Common Applications Kept Enhanced (CAKE) allow to specify the maximum packet dequeue rate.

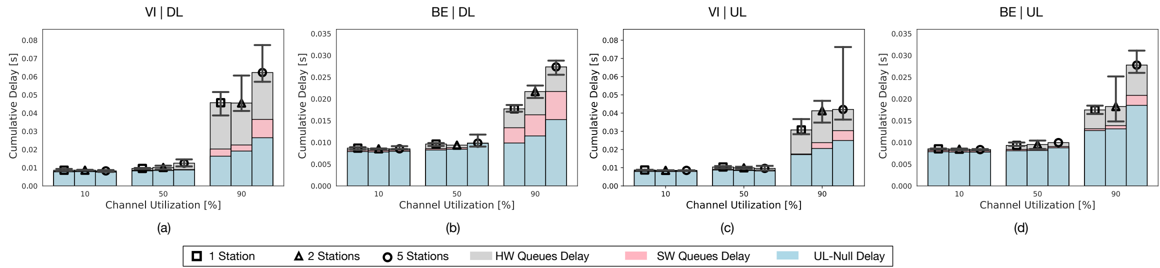

Fig. 4.3: Delay components of wired-to-wireless switching delay in the presence of concurrent traffic. (a) Downlink (DL) video concurrent traffic. (b) Downlink (DL) best-effort concurrent traffic. (c) Uplink (UL) video concurrent traffic. (d) Uplink (UL) best-effort concurrent traffic.

queues. Four of these queues correspond to the voice, video, best-effort, and background access categories, and the last queue is used for management and control packets. Each queue is associated with a Distributed Coordination Function (DCF) unit that contends for channel access according to the Enhanced Distributed Channel Access (EDCA) parameters specified by the 802.11e amendment. These contention parameters increase the probability of longer wait times for lower access category queues, thereby prioritizing higher access categories. The voice and background access categories have the highest and lowest priorities. Each individual DCF unit contends with other stations, as well as the DCF queues from the same station. The latter contention is known as internal or virtual collision. In case of internal collision, the higher-priority access category is allowed to access the channel. Once a HW Queue obtains access to the channel for transmission, it transmits the packet to the station and generates an interrupt that triggers the dequeing of packets from the SW Queues.

4.3.3 Empirical Evaluation of Wired-to-wireless Switching Delay

In this section, we use the FLIP framework for monitoring the delays across the stages of the WiFi networking stack.

The testbed setup includes an IoT station that utilizes the APSM energy ef-

ficiency mechanism. The station is associated with a FLIP AP. The testbed includes additional stations to introduce concurrent network traffic. The intensity of the traffic generated by these stations is represented as *CU* in the results. CU is defined as $d_{activity}/d_{overall}$, where $d_{activity}$ is the time duration the radio sensed a signal higher than a pre-specified threshold value during time duration $d_{overall}$. A server (connected to the AP via the wired interface) sends a ping packet to the IoT station per second. This packet belongs to the best-effort access category. We refer to this traffic as the *IoT traffic*. We define *cumulative delay* as the duration between the time instance a packet arrives at the W-NIC of the AP and the time instance it is successfully transmitted over the WL-NIC. Referring to Figure 4.2, we also monitor delay components $\delta_a, \delta_b, \delta_c, \delta_d, \delta_e$, defined as follows:

– δ_a: The time spent by a packet in qdisc,

– δ_b The interval between the insertion of a packet into the SW Queues and the next beacon announcement,

– δ_c: The time duration between the beacon announcement with TIM bit set and the NULL packet received by the AP,

– δ_d: The delay incurred in mac80211's SW Queues, when the HW Queues are full,

– δ_e: The amount of time spent in the HW Queues when contending for channel access.

It is worth noting that with the recent improvements in Linux wireless networking, some (new) drivers bypass the qdisc module. The ath9k driver used by the AP in our testbed implements this approach. Since we observed that δ_a was always less than 1 ms, we do not show δ_a in the empirical evaluation results of this chapter. The duration between the time instance the packet was enqueued in the SW Queues and the beacon

announcement with the station's TIM bit set (i.e., δ_b) depends on the packet arrival time.

Figure 4.3 presents the components of wired-to-wireless switching delay in the presence of various channel utilization levels. We consider the impact of different types of channel utilization:

– Traffic direction: (i) DL: when the traffic direction is from AP to the stations, and (ii) UL: when the traffic direction is from stations to the AP,

– We consider the two access categories: video and best effort.

Our results show that the cumulative delay of IoT traffic increases as CU intensifies. However, we observed that for a particular CU intensity, the values of packet delivery delays vary depending on the access category and direction of the concurrent traffic. As Figure 4.3(b) shows, the median cumulative delay in the presence of 90% best-effort CU is 27 ms, whereas, as the results of Figure 4.3(a) suggest, the cumulative delay with 90% video CU is 61 ms (i.e., 125% higher). This behavior is justified by the 802.11e amendment which specifies the channel access contention parameters for the four access categories. Compared to the best-effort access category, the 802.11e amendment allows faster, more probable access of video HW Queue to the channel. Additionally, every time the video HW Queue grabs the channel, it is allows to continuously use the channel for 3.008 ms; whereas, the best-effort HW Queue can send one packet per channel access. Because of these two reasons, IoT traffic (best-effort) spends more time in the HW Queue when competing with video traffic. When the concurrent traffic type is best-effort too, both traffic get equal access to the channel and delay drops. However, once the best-effort HW Queue is full due to the higher traffic rate of concurrent traffic, dequeing of packets from SW Queues is halted. Thus, the waiting times of the packet in the best-effort SW Queues increase as well. This is observed in Figure 4.3 (b) as we

increase CU. The waiting duration in the SW Queue contributes to about 10% of the cumulative delay, when best-effort concurrent traffic consumes 90% of the channel.

We also observe that δ_c accounts for a significant portion of the total delay. Specifically, as channel contention due to UL or DL traffic escalates, this delay increases too. This is because Null packets are regular data packets belonging to best-effort access category, and therefore, they need to contend with concurrent traffic. Additionally, even when the CU level is low, Null packets are sent at least 7 ms after the beacon. We observed that this is due to the guard times employed by the station around in each beacon reception instance.

When the concurrent traffic is UL (Figures 4.3 (c) and (d)), increasing the number of stations results in a higher delay of IoT traffic. From the CSMA point of view, this is because increasing the number of stations reduces the chance of winning the channel by the IoT station. However, we observe a similar trend when the concurrent traffic is DL and the AP is the only device transmitting packets. As explained in Section 4.3.2.3, the stations compete internally to gain channel access; therefore, increasing the number of stations receiving concurrent traffic reduces the chance of channel access for IoT traffic.

Discussion. In this section we assumed that the IoT station wake up at all beacon instances. However, in [93] we showed that stations can skip some beacons instances and lower the overhead of beacon reception. This is achieved by using a listen interval value, denoted as τ, that denotes the number of beacons skipped between wake-ups. For $\tau > 1$, the time spent in SW Queues is highly affected. For a station using listen interval τ, assume the beacon instances during the listen interval are $[t_k, ..., t_{k+\tau}]$. When a ping destined to the IoT station arrives arrives at the AP's W-NIC during interval $[t_k, t_{k+1})$, τ beacon packets must be sent before the next wake-up instance of the station. Similarly, if the packet arrives during interval $[t_{k+1}, t_{k+2})$, the AP needs to

send $\tau - 1$ beacon packets. Therefore, the expected number of beacons sent until station wake-up is $\frac{1}{\tau}(\tau + (\tau - 1) + ... + 2 + 1) = (\tau + 1)/2$, when $\tau > 1$. The expected wake up delay is computed as $102.4 \times (\tau + 1)/2$ ms and maximum wake up delay is $\tau \times 102.4$ ms, for $\tau > 1$.

4.4 Passive Monitoring of Stations' Energy Consumption

In this section, we present a novel method to passively track the duty cycle and energy consumption of stations by the AP. Specifically, instead of using additional hardware to measure the energy consumption of stations, we collect the duty cycle pattern of stations from their associated AP's driver.

4.4.1 FLIP's Methodology for Monitoring the Energy Consumption of Stations

To monitor the duty cycle of stations, we rely on the observation that stations inform the AP whenever they transition to another mode (sleep to awake and vice-versa). With PSM, the station sends a PS-Poll packet to the AP to express its transition into awake mode. The station retrieves queued packets until the *more-data* field is set to '0' in the data packet sent by the AP, and then transitions into sleep mode. With APSM, the station can wake up or transition into sleep mode anytime. The station informs the AP about these transitions by the power-save-bit ('0': waking up, '1': transitioning into sleep state) inside Null packets.

Within the FLIP framework, NSM is capable of keeping track of the timestamp, type, sub-type, direction, and the power-save-bit of each packets exchanged with the AP. To reduce processing overhead, the NSM-K module processes the packets belonging to

the stations whose energy are being monitored. The set of these stations are programmed using the NSM-U module. Hence, for each station, via the power-save-bit and more-data fields, NSM logs the instances the station changes its operational mode. This allows FLIP to keep track of the wake-up duration of the station.

It must be noted that, stations do not inform the AP when they wake up for beacon reception; therefore, the duty cycle pattern inferred by the approach explained above does not include the overhead of beacon reception. In order to track the station's wake-up instances for beacon reception, we rely on the information provided by `hostapd`, `mac80211`, and driver. Expiration of beacon alert timer (a.k.a., `bcntimer`) in the WL-NIC generates an interrupt for beacon transmission. This interrupt is handled by the `ieee80211_beacon_get` function, which generates and transmits a beacon. We probe this function to keep track of the timestamps and the number of beacons sent during the monitoring duration. However, the stations may not wake up at every beacon instance. As explained in Section 4.3.3, the stations may specify a listen interval value to reduce the overhead of beacon reception. The listen interval value is informed by the station to the AP during the association process; this value is maintained by AP's `hostapd` module (cf. Figure 4.1). NSM-U utilizes the `hostapd` logs to obtain the listen interval value for each associated station. With the number of beacons sent during the monitoring period and the listen interval of the station, the NSM-U module calculates the number of times the station woke up to receive beacons. To accommodate for time synchronization inaccuracy, stations allocate guard awake times around beacon reception instances[3]. In this section, we assume the station's wake-up duration per beacon instance is d_b. The total duration spent in awake mode by the station is calculated by NSM-U as $\Psi = (d_b \times c_b) + \psi$, where c_b denotes number of beacons during the monitoring period and and ψ is station awake time inferred from power-save-bit and more-data fields. The

[3]This was also observed in Section 4.3.3, where the UL Null packet sent by the station was at least 7 ms after beacon announcement. A thorough study of beacon reception overhead can be found in [93].

duty cycle during an interval $[t_m, t_n]$ is computed as $\mathcal{D} = \frac{\Psi}{t_n - t_m}$.

To calculate the energy consumed by the WL-NIC, its various operational modes must be considered: (i) sleep, (ii) reception: when the device is receiving packets, (iii) idle (a.k.a., idle listening): when the device is ready to receive packets, and (iv) transmission: when the device is transmitting packet. Devices' data-sheets provide the power consumption of these operational modes. Power consumption of idle and reception modes are very similar, and we denote their power consumption as p_{rx}. The energy consumed during idle and reception modes is calculated as $p_{rx} \times (\Psi - \Psi_{tx})$. To compute the energy consumed in transmission mode, we need to extract the time spent by the station transmitting packets. This time is calculated as $\Psi_{tx} = \sum_{\forall p \in \mathbf{P}} \frac{l(p)}{r_p}$, where $\mathbf{P}$ is the set of packets sent by the station during the monitoring duration, l_p is the length (bits) of packet p, and r_p is the physical-layer transmission rate (bps) of packet p. Within the FLIP framework, NSM-U extracts the data rate and the length of the received packets from the driver. For example, ath9k maintains the data rate and size of received packets inside the `rs_rate` and `rs_datalen` fields within the `ath_rx_status` structure. We calculate the energy consumption during a monitoring period $[t_m, t_n]$ as follows: $\mathcal{E} = p_{rx} \times (\Psi - \Psi_{tx}) + p_{tx} \times \Psi_{tx} + p_{sleep} \times (t_n - t_m - \Psi)$, where p_{tx} is the power consumption in transmission mode, and p_{sleep} is the power consumption of sleep mode.

4.4.2 Empirical Evaluation of the Accuracy of Passive Energy Monitoring

We validate the accuracy of FLIP for measuring the duty cycle and energy consumption of stations' WL-NIC. We compare passive energy mounting against the results collected from a commercial energy monitoring tool [94]. In the first set of experiments, we use CYW43907 [7], which is a low-power, RTOS-based 802.11n SoC

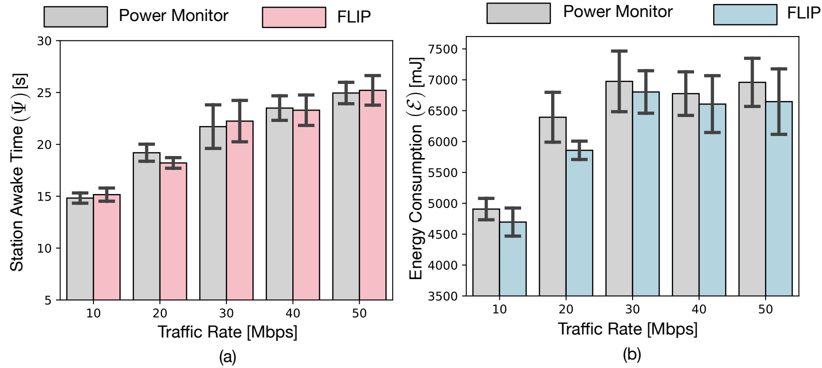

Fig. 4.4: Comparison of FLIP's passive energy monitoring versus the energy consumption measured by a commercial power monitoring tool. The x-axis is the incoming data rate of the station. (a) Awake time (Ψ) and (b) energy consumption ($\mathcal{E}$). The station used is CYW43907 operating on APSM with tail-time set to 10ms.

designed for IoT applications. The physical layer communication rate between this station and the AP is 54 Mbps. We use iperf to exchange traffic with the station. The station uses the APSM energy efficiency mechanism with its tail-time set to 10 ms. Figures 4.4 (a) and (b) compare the awake time and energy consumption results, respectively. Each experiment is 30 seconds long, each point in the graph is the median of ten experiments, and error bars represent lower and higher quartiles. These results confirm that the measurement error of FLIP is within 6% of the baseline.

In the second set of experiments, in addition to the CYW43907 station, we use WLE900VX [95] which relies on a Linux-based driver. We observed that WLE900VX does not support tail time below 50 ms, hence, we changed the tail of CYW43907 to 50 ms too. Also, we vary the experiments' duration to validate the accuracy of passive monitoring for various experimentation intervals. To this end, we send a ping packet to the station per second, and vary the total number of pings sent during each experiment. Figure 4.5 presents the results. These results confirm that the measurement error of FLIP is within 9% of the baseline for experiments as long as 500 seconds.

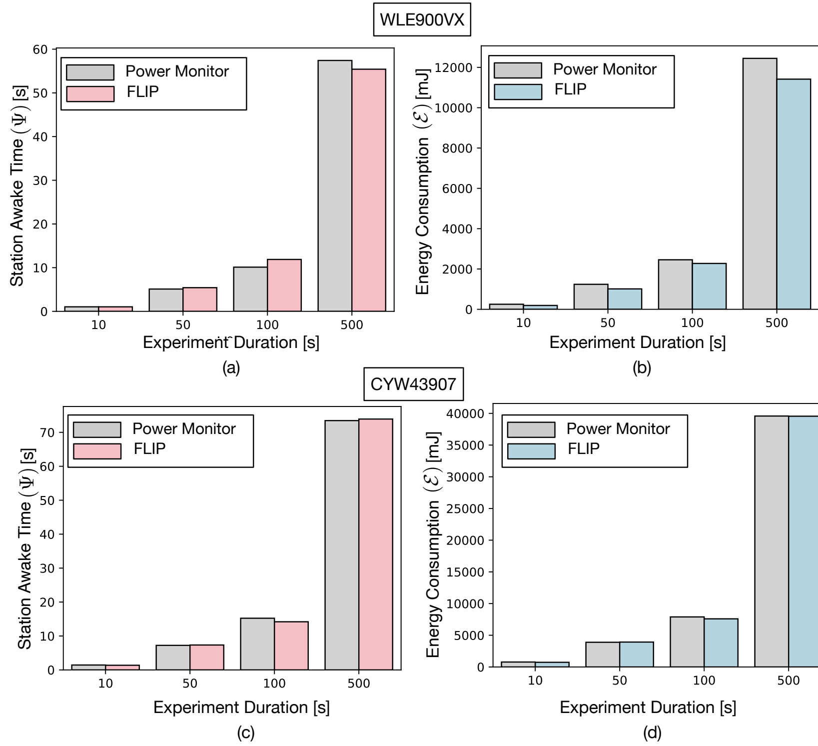

Fig. 4.5: Awake time (Ψ) and energy consumption ($\mathcal{E}$) of stations measured by FLIP and a commercial power monitoring tool. The x-axis represents the experiment duration, which also corresponds with the number of ping packets received by the station. (a) and (b): The station is CYW43907. (c) and (d): The station is WLE900VX. For both stations the tail time of APSD is set to 50 ms.

4.5 Summary

In this chapter, we studied the internals of the WiFi networking stack and demonstrated how various components of this stack handle the operations pertaining to packet switching at APs and energy-efficient operation of stations. We then leveraged eBPF to augment WiFi APs and performed system monitoring in terms of packet switching delay and tracking the energy consumption of stations. The empirical studies of this chapter show how the proposed framework can be used to investigate the operation of WiFi networks. In addition to investigating various aspects of WiFi networks, the FLIP framework can also be used to collect monitoring data and develop methods that react to network dynamics. For example, by providing insights into packet

switching delay and the instantaneous energy efficiency of stations, FLIP facilitates the development of algorithms that manage the steering of stations among APs. Developing methods that rely on the FLIP framework is left as future work.

This page is intentionally left blank

Chapter 5

Enhancing the Energy-Efficiency and Timeliness of IoT Communication in WiFi Networks

5.1 Introduction

Contention between the flows of IoT stations as well as the contention between these flows and regular user-generated traffic for downlink packet transmission reduces the energy efficiency and timeliness of IoT communication. In this chapter, we propose *Wiotap* (WiFi IoT access point) to address the problem of joint channel access and power saving in 802.11-based networks. We assume that the network includes user devices (such as smartphones and laptops) as well as IoT devices (such as battery powered medical monitoring devices). For networks with a large number of IoT stations, Wiotap significantly enhances energy efficiency by applying per-packet scheduling. In networks, consisting of both regular and IoT traffic, Wiotap ensures regular traffic does not impact the energy efficiency of IoT stations. The proposed system includes a least-laxity first (LLF) packet scheduling mechanism based on the tolerable delay of each packet before the expiry of the destination station's tail time. The proposed mechanism introduces dedicated queues for IoT traffic and allocates deadline guarantees to each queue based on the distribution of tolerable delay values collected during a time window. Once a packet arrives on the access point's wired interface, if the destination station is awake, the access point assigns a priority to the packet and pushes it into the selected IoT queue.

The assigned priority depends on the deadline of this packet relative to the deadline of all the awake IoT stations. This mechanism reduces the waiting time of IoT stations and increases the number of packets exchanged during a wake-up period. If the access point determines that a packet cannot be delivered to its destination station before its transition into sleep mode, a priority that is higher than that of regular stations' packets is assigned to the IoT packet. This mechanism expedites the delivery of packets to IoT stations after reception of beacon packets during the next wake up period. To enforce the delivery delay guarantee of queues, we employ a time-division-based traffic shaping approach to control the service rate of the queues. Specifically, in this mechanism, time is divided into service intervals, and the maximum number of packets serviced by each queue during this interval is limited.

We evaluate the performance of the proposed mechanism using simulation and testbed. In particular, since Wiotap's per-packet scheduling is especially applicable to large-scale deployments, we had to use simulation to demonstrate system scalability. Based on the simulation results, compared to a regular AP, our solution provides an average performance improvement of 37%. Additionally, increasing the number of queues dedicated to IoT traffic from 2 to 4 can further reduce the duty cycle of IoT stations by 28%.

To confirm the implementation accuracy and performance benefits of Wiotap even when the number of IoT stations is not high, we have implemented a testbed using four IoT stations and different types of regular traffic generators. To show the impact of server location on performance, we have changed the location of MQTT to represent edge and cloud scenarios. The empirical results indicate that Wiotap reduces delay and energy consumption by up to 52% and 44% in the edge scenario and 41% and 38% in the cloud scenario.

The remainder of this chapter is organized as follows: Section 5.2 introduces

power-save modes, traffic prioritization in 802.11 networks, and the employed system model. Section 5.3 presents the proposed approach. The implementation details of the proposed approach are explained in Section 5.4. Sections 5.5 and 5.6 present simulation and empirical performance evaluations. Finally, Section 5.7 summarizes the chapter and presents future directions.

5.2 System Overview

In this chapter, we first present an overview of 802.11 in terms traffic scheduling and traffic prioritization. We then present the system overview of EAPS.

5.2.1 Traffic Prioritization

Since 802.11 networks are used for the exchange of both elastic and real-time data, the 802.11e standard provides various access categories (AC) [51]. The AC of MAC frames is determined based on the differentiated services code point (DSCP) field[1] of IP header. This field comprises of different flagged bits, which when set, conveys to the lower layers the flow type and enforces IP precedence for per-hop QoS and priority. This layer-3 field is then mapped to the class of service (CoS) field in the MAC header [52]. By using CoS mapping, the kernel sets the priority socket buffer and enqueues the packet to the corresponding transmit queue. In the 802.11e protocol, each EDCA queue behaves as a virtual station and contends for the channel independently according to the contention parameters specified by the 802.11e standard.

Additionally, almost all Linux systems are capable of administering the manner in which the packets are sent over a medium by employing queueing disciplines known

[1]This field is also know as the type of service (ToS).

as *qdisc*. Situated between layer-2 and layer-3 of IP stack, qdisc provides several types of traffic scheduling (what packet to be sent) and traffic shaping (how many packets to be sent per time unit) mechanisms to hand packets over to layer-2.

5.2.2 System Model

The system is composed of an *access point* (AP) and *stations*. There are two types of stations: (i) *regular stations*, denoted by s_i^{reg}, such as smartphones and laptops, and (ii) *IoT stations*, denoted by s_i^{iot}, such as medical IoT devices or industrial robot arms which perform machine-to-machine communication. Although regular stations might exchange voice and video traffic with the AP, we assume that IoT stations are resource-constrained and therefore, preserving their energy resources is more important compared to regular stations. In addition, even if the network only includes IoT stations, we are interested in reducing the energy consumption of all the stations. Once a station exchanges a packet with the AP, it stays awake for a tail time duration Γ.

5.3 Scheduling Mechanism

Figure 5.1 shows the queue allocation scheme. On top of the MAC layer, we introduce *IoT queues* in addition to the *regular* queues, in the qdisc layer. Once a packet for a regular (non-IoT station) arrives on the wired interface of AP, it is pushed into one of the regular queues, i.e., $\mathbf{Q}_{net}^{reg} = \{Q_{vo}, Q_{vi}, Q_{be}, Q_{bk}\}$, depending on the packet's TOS field. Packets destined to IoT stations are pushed into one of the IoT queues, i.e., $\mathbf{Q}_{net}^{iot} = \{Q_0, Q_1, ..., Q_{\eta-1}\}$, to accelerate the transmission of these packets. We refer to Q_0 as the *base queue* and Q_1 to $Q_{\eta-1}$ as the *prioritized queues*. The queue selection process is based on a scheduling algorithm that we will present later on. The qdisc

packets are assigned to the MAC layer queues, i.e., $\mathbf{Q}_{mac} = \{Q'_{vo}, Q'_{vi}, Q'_{be}, Q'_{bk}\}$, based on their priority, as indicated by the arrows in Figure 5.1.

The basic idea of transmission scheduling is to prioritize the packets of IoT stations to reduce their idle listening time and the number of sleep-awake transitions. When a packet belonging to an IoT station arrives, if the station is awake, we evaluate the time left for that station to return to the sleep mode. If the remaining duration is longer than a threshold, then the packet is accelerated for delivery. In addition, the acceleration algorithm tries to maximize the chance of packet delivery to all the IoT stations by taking into account the relative priority of all the packets' deadlines. We explain these operations in the rest of this section.

5.3.1 Acceleration Eligibility

A packet is eligible for acceleration if its destination IoT station is still awake and acceleration results in a packet delivery before the end of the tail time. To this end, it is essential to keep track of the operational status of all IoT stations and determine the remaining tail time of awake stations. To determine the remaining duration, however, it is not possible to simply rely on the sleep/awake schedule of stations because the tail time is renewed every time a packet is exchanged with the AP. Therefore, it is necessary to record the time stamp of the last packet exchanged with the AP. Section 5.4 will present the implementation details.

Assume a packet p_j belonging to an IoT station s_i^{iot} arrives at time t. This packet is eligible for acceleration if:

$$\Delta(p_j) = \Gamma(s_i^{iot}) - (t - \tilde{t}(s_i^{iot})) > Th(s_i^{iot}) \tag{5.1}$$

where $\Delta(p_j)$ reflects the *delivery laxity* of packet p_j upon its arrival, $\Gamma(s_i^{iot})$ is the tail

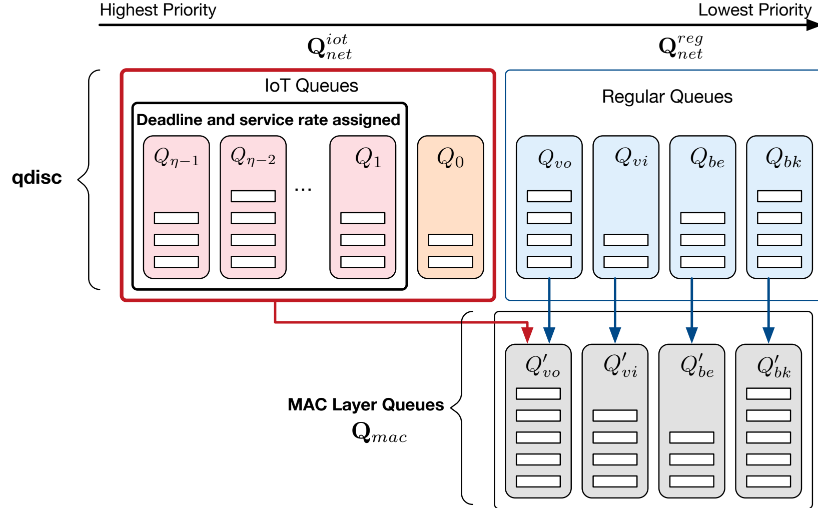

Fig. 5.1: The proposed scheduling algorithm introduces IoT queues in addition to the regular queues in the qdisc layer. The IoT queues are configured based on the delay distribution of IoT packets.

time of the station, t is the current time, and $\tilde{t}(s_i^{iot})$ is the last time the station has exchanged a packet with the AP. In addition to decision making about the acceleration of incoming packets, the computed Δ values are also used for queue configuration. We will present *Queue Configuration* algorithm in Section 5.3.2. If Inequality 5.1 holds for an incoming packet, then the *Enqueue* algorithm tries to accelerate the transmission of this packet by determining a suitable prioritized queue within the set $\{Q_1, Q_2, ..., Q_{\eta-1}\}$. We will present this algorithm in Section 5.3.3. If Inequality 5.1 does not hold, then the packet is simply pushed into the *base queue* Q_0 to increase its priority compared to regular traffic. The base queue expedites the delivery of this packet compared to regular traffic during the next wake up time.

The threshold value $Th(s_i^{iot})$ depends on channel access contention, physical layer rate, link reliability between the AP and the station, and the rate of regular traffic. For example, if multiple transmissions are required to reach the station, then the threshold value should be long enough to account for the retransmissions. Since all the qdisc IoT queues are mapped to MAC layer's queue Q'_{vo}, to measure these delays,

we record the time interval between the instance an IoT packet arrives in Q'_{vo} until its successful delivery to the destination node. These values are stored in a circular array **Tx**.

5.3.2 Queue Configuration

In this section we present the operation of Queue Configuration algorithm, which assigns priorities to the IoT queues to enable fine-grained prioritization of IoT packets.

We denote the set of queues devoted to IoT packets as $\mathbf{Q}_{net}^{iot} = \{Q_0, Q_1, ..., Q_{\eta-1}\}$. While queue Q_0 is used to simply prioritize IoT packets that are not acceleration eligible, the rest of the queues are configured by the Queue Configuration algorithm to offer deadline-aware acceleration. Associated with each prioritized queue $Q_i \in \{Q_1, ..., Q_{\eta-1}\}$ is a *maximum tolerable delay* (MTD) value, denoted as $\mathcal{M}(Q_i)$. The MTD of a queue reflects the maximum delay experienced by the packets of that queue until transmission. This is to ensure that the deadline of the packets buffered in that queue satisfy their delivery deadline requirement.

The purpose of queue configuration is to assign the MTD of the IoT queues based on the distribution of packet laxities computed by Equation 5.1. Whenever a new Δ value is computed for an incoming IoT packet, the value is inserted into a circular queue denoted as $\mathbf{\Delta}$. The main idea is to divide the range of deadlines into two equal intervals and configure the MTD of queues based on the number of deadline entries that fall in each interval. Each interval is then broken into two more intervals, and the same process is repeated until MTD values are assigned to all the queues.

The Queue Configuration algorithm is presented in Algorithm 1. The high level function `queue_conf()` computes the minimum and maximum of the laxity values

stored in the circular queue $\boldsymbol{\Delta}$. These values, in addition to the minimum and maximum index of the IoT queues, are passed to function `qc()`, which is recursively called to assign MTDs to queues. This function first computes the mid point of laxity values in the range $[\Delta_{min}, \Delta_{max}]$. The number of queues assigned to the left and right side of the queue correspond to the distribution of laxity values around the mid value. If the number of queues assigned to the left side is equal to one, then the MTD of that queue is equal to the Δ_{mid} of that iteration. If the number of assigned queues is more than one, then the function is recursively called to configure left side queues. Queue allocation to the right side is performed in a similar manner. If the number of queues assigned to the right side is equal to one, then the MTD of that queue is equal to the Δ_{max} of that iteration. The time complexity of this algorithm is $O(n)$ because the division process continues until all the queues are configured individually.

Algorithm 1: Queue Configuration Algorithm

1. **function** `queue_conf()`
2. $\quad \boldsymbol{\Delta} = \{\Delta_i\}_{i=1}^{N}$;
3. $\quad \Delta_{min} = \min_i \Delta_i$;
4. $\quad \Delta_{max} = \max_i \Delta_i$;
5. $\quad I_{max} = \eta - 1$;
6. $\quad I_{min} = 0$;
7. $\quad$ `qa`$(\Delta_{min}, \Delta_{max}, I_{min}, I_{max})$;
8. **function** `qc`$(\Delta_{min}, \Delta_{max}, I_{min}, I_{max})$
9. $\quad \Delta_{mid} = \Delta_{min} + (\Delta_{max} - \Delta_{min})/2$;
10. $\quad q = I_{max} - I_{min} + 1$;
11. $\quad W_{left} = 0$;
12. $\quad W_{right} = 0$;
13. $\quad$ **for** *every entry* Δ_i *in* $\boldsymbol{\Delta}$ **do**
14. $\quad\quad$ **if** $\Delta_i \geq \Delta_{min}$ and $\Delta_i \leq \Delta_{mid}$ **then**
15. $\quad\quad\quad W_{left} + +$;
16. $\quad\quad$ **if** $\Delta_i > \Delta_{mid}$ and $\Delta_i \leq \Delta_{max}$ **then**
17. $\quad\quad\quad W_{right} + +$;
18. $\quad W_{total} = W_{left} + W_{right}$;
19. $\quad W_{left} = W_{left}/W_{total}$;
20. $\quad W_{right} = W_{right}/W_{total}$;
21. $\quad W'_{left} = W_{left} \times q$;
22. $\quad W'_{right} = W_{right} \times q$;
23. $\quad W'_{left} = \left\lfloor W'_{left} + 0.5 \right\rfloor$;
24. $\quad W'_{right} = \left\lfloor W'_{right} + 0.5 \right\rfloor$;
25. $\quad$ **if** $W'_{left} + W'_{right} == q + 1$ **then**
26. $\quad\quad$ randomly reduce the number of left or right queues by one;
27. $\quad$ `/*allocate queue to the values less than or equal to` Δ_{mid}`/*`
28. $\quad$ **if** $W'_{left} == 1$ **then**
29. $\quad\quad \mathcal{M}(Q_{\eta-1}) = \Delta_{mid}$;
30. $\quad$ **else if** $W'_{left} > 1$ **then**
31. $\quad\quad left_end = \eta - 1$;
32. $\quad\quad left_start = \eta - W'_{left} + 1$;
33. $\quad\quad$ `qc`$(\Delta_{min}, \Delta_{mid}, left_start, left_end)$;
34. $\quad$ `/*allocate queue to the values greater than` Δ_{mid}`/*`
35. $\quad$ **if** $W'_{right} == 1$ **then**
36. $\quad\quad \mathcal{M}(Q_{I_{min}}) = \Delta_{max}$;
37. $\quad$ **else if** $W'_{right} > 1$ **then**
38. $\quad\quad left_start = \eta - W'_{left}$;
39. $\quad\quad right_end = left_start - 1$;
40. $\quad\quad right_start = left_start - right_queues$;
41. $\quad\quad$ `qc`$(\Delta_{mid}, \Delta_{max}, right_start, right_end)$;

5.3.3 Enqueue Algorithm

A packet p_j satisfying Inequality 5.1 might be assigned to a prioritized IoT queue if the following condition is satisfied,

$$\exists i \in [1, \eta - 1] \;\mid\; \Delta(p_j) \leq \mathcal{M}(Q_i). \tag{5.2}$$

If none of the IoT queues can satisfy the above condition, then the packet is inserted into the base IoT queue Q_0. If Condition 5.2 is satisfied, then a least-laxity first (LLF) scheduling strategy is employed to assign packets to the prioritized queues. To this end, the packet is added to the queue with the highest MTD value that can satisfy the delivery deadline. In other words, the index of the eligible queue, denoted as i, is found as follows:

$$\underset{i \in [1, \eta-1]}{\operatorname{argmin}} \big(\Delta(p_j) \leq \mathcal{M}(Q_i)\big). \tag{5.3}$$

However, it should be noted that the deadline of a queue does not only reflect the transmission duration of the packets in that queue. Instead, for each Q_i, its deadline is the deadline of its next higher priority queue (i.e., Q_{i+1}) plus the duration required to transmit the packets in Q_i. Therefore, to guarantee a maximum transmission delay for the packets of each queue, the following inequality must be true,

$$\mathcal{M}(Q_{i+1}) + \mathcal{D}(Q_i) < \mathcal{M}(Q_i), \quad 1 < i < \eta - 2 \tag{5.4}$$

where $\mathcal{D}(Q_j)$ is the delay of transmitting the packets in queue Q_j. However, to ensure the validity of the above inequality, it is essential to limit the number of packets serviced by each queue per time unit. We employ a time-division-based mechanism to satisfy

this requirement. For each queue $i \in [1, \eta - 1]$ we define its *service capacity* as

$$\bar{\mathcal{S}}(Q_i) = \lfloor (\mathcal{M}(Q_i) - \mathcal{M}(Q_{i+1}))/\mu_{\mathbf{Tx}} \rfloor \tag{5.5}$$

where $\mu_{\mathbf{Tx}}$ is the average of the values stored in the circular array **Tx**. In other words, service capacity represents the number of serviceable packets per *service period*. Since for the highest priority queue

$$\bar{\mathcal{S}}(Q_{\eta-1}) = \lfloor \mathcal{M}(Q_{\eta-1})/\mu_{\mathbf{Tx}} \rfloor \,, \tag{5.6}$$

service period is defined as the MTD of the lowest priority queue, i.e., $\mathcal{M}(Q_1)$. In order to enforce service capacities, in addition to the $\bar{\mathcal{S}}(Q_i)$ values assigned to each queue, the number of packets that have been added to each queue during the current service period is maintained by the Enqueue algorithm. For each queue Q_i this value is denoted by $\mathcal{S}(Q_i)$. No more packets are inserted into a queue Q_i during the current service period if $\mathcal{S}(Q_i) \geq \bar{\mathcal{S}}(Q_i)$. The $\mathcal{S}(Q_i)$ counters are reset at the beginning of each $\mathcal{M}(Q_1)$ interval.

Enforcing service capacity imposes another limitation on finding an appropriate queue for an acceleration eligible IoT packet. In other words, it is possible that the MTD of a packet satisfies the incoming packet's delivery deadline, but the queue has exceeded the maximum number of allowable insertions in the current service period. More formally, the index i returned by formula 5.3 does not satisfy Equation 5.5. In this case, the search for an appropriate queue continues towards higher priority queues. Therefore, to ensure a sustained guarantee of deliver deadlines, Condition 5.3 must be revised as follows,

$$\underset{i \in [1, \eta-1]}{\operatorname{argmin}} \big(\Delta(p_j) \leq \mathcal{M}(Q_i) \;\text{ and }\; \mathcal{S}(Q_i) < \bar{\mathcal{S}}(Q_i) \big). \tag{5.7}$$

Algorithm 2 shows the packet enqueue process. This algorithm tries to find a prioritized queue for the incoming packet if the packet deadline is higher than the threshold we discussed in Inequality 5.1 (cf. Line 5). The algorithm then pushes the packet into the base queue if the packet deadline is longer than the MTD of the lowest priority queue (cf. Line 6). Please note that the service capacity of this queue is not taken into account since there is no deadline guarantee offered by this queue. If the packet has not been inserted into the base queue, then the algorithm verifies if the deadline is lower than that of the highest priority queue. In this case, the packet is pushed into the queue if the queue is capable of serving more packets during the current service period (cf. Line 10). Otherwise, the packet is inserted into the base queue (cf. Line 14) since it is evident that the other queues cannot satisfy the deadline requirement of this packet. If none of the above two boundary cases hold, the algorithm tries to find a queue that satisfies the deadline requirement and is capable of serving more packets during the current service period (cf. Line 18). Please note that, since the packet service rate of each queue is limited, after the completion of this loop, the algorithm dos not necessarily push the packet into the lowest priority queue that can satisfy the packet deadline. In the end, if no prioritized queue was found, the packet is inserted into the base queue. The time complexity of this algorithm is $O(n)$ because if the boundary values do not hold the algorithm need to evaluate the eligibility of all the queues.

5.4 Implementation

Figure 5.2 presents the implementation architecture of Wiotap in Linux-based APs. The implementation is composed of four main modules: (i) *Scheduler module*, which includes a kernel-space sub-module (S-KNL), and a user-space sub-module (S-USL), (ii) *WiFi Logger* (WiLog) module, and (iii) *qdisc module*.

Algorithm 2: Enqueue Algorithm

```
1  function enqueue(p_i)
2      η = |Q|;
3      s_i^iot is the destination of packet p_j;
4      Δ(p_j) = Γ(s_i^iot) − (t − t̃(s_i^iot));
5      if Δ(p_j) > Th(s_i^iot) then
6          if Δ(p_j) > M(Q_1) then
7              insert p_j into Q_0;
8              return;
9          else if Δ(p_j) ≤ M(Q_{η−1}) then
10             if S(Q_{η−1}) < S̄(Q_{η−1}) then
11                 S(Q_{η−1}) + +;
12                 insert p_j into Q_{η−1};
13                 return;
14             else
15                 insert p_j into Q_0;
16                 return;
17         else
18             for k = 1 to η − 1 do
19                 if Δ(p_j) < M(Q_k) then
20                     if S(Q_k) < S̄(Q_k) then
21                         S(Q_k) + +;
22                         insert p_j into Q_k;
23                         return;
24     insert p_j into Q_0;
25     return;
```

The S-USL module runs the Queue Configuration and Enqueue algorithms. The S-KNL module modifies the TOS field in the IP header of IoT packets according to the queue number specified by S-USL. The WiLog module keeps track of the operational status of IoT stations. Finally, the TOS field set by the S-KNL module is used by the qdisc kernel module to push the packet into the appropriate queue.

The Wiotap system also includes the `hostapd` daemon [96], which is a user-space software capable of enabling a WiFi radio as an AP. This daemon communicates

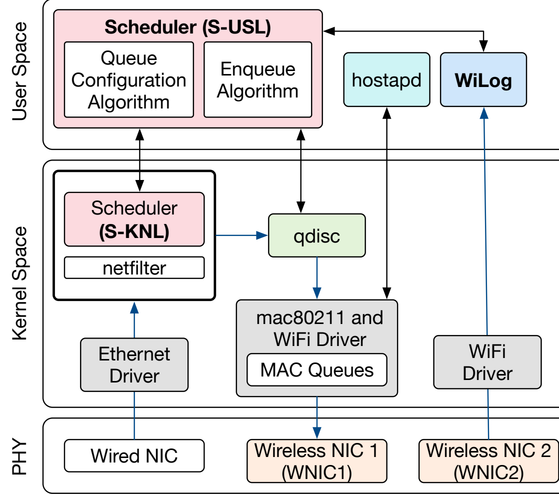

Fig. 5.2: Implementation architecture of the proposed Wiotap access point.

with `cfg80211` through `nl80211`.

5.4.1 WiFi Logger Module (WiLog)

The proposed scheduling mechanism requires knowledge about the current status of IoT stations regarding their sleep/awake status and the remaining tail duration of awake stations. The WiLog module is responsible for providing the S-USL module with this information. To this end, Wiotap is equipped with one additional wireless NIC to overhear all the packets and collect the required information. Specifically, the WiLog module keeps track of both data and NULL packet exchange activities. Also, this module includes the circular array **Tx** explained in Section 5.3.1. The capacity of this array is 100 in our implementation.

5.4.2 Scheduler Module

The scheduler is implemented in two parts: user space and kernel space, which are referred to as S-USL and S-KNL modules, respectively. To avoid floating-point

calculations in the kernel space [97], the S-USL module, which includes the Queue Configuration and Enqueue algorithms, is implemented in the user space. The S-KNL module uses the netfilter [98] components located in the Linux kernel to grab the packets arriving on the Ethernet interface. Specifically, we use the `PREROUTING` mode, where all the packets are intercepted before the routing decision has been made. If the destination of an arriving packet is an IoT station, the S-KNL module requests the S-USL module to determine the most appropriate queue for this packet. S-USL collects $\tilde{t}(s_i^{iot})$ and $\mu_{\mathbf{Tx}}$ from the WiLog module. After determining a queue, S-USL instructs S-KNL to modify the IP header of the packet to reflect the new TOS value determined. After modifying the IP header, we recalculate the checksum and pass the packet to the qdisc module to place it in the proper queue. The circular queue $\mathbf{\Delta}$ used to hold the packet laxity values is implemented in the S-USL module. The size of this queue in our implementation is 100.

5.4.3 qdisc Module

By default, every network interface is assigned a `pfifo_fast` as its transmit queuing mechanism [99, 100, 101]. `pfifo_fast` implements a simple three-band prioritization scheme based on the 8-bit TOS field in the IP header [102]. Within each band, FIFO rules apply. However, as long as packets are waiting in band 0, band 1 cannot be processed. A similar policy is applied to band 1 and band 2. Hence, `pfifo_fast` always processes band-0's traffic regardless of the number and rates of contending flows [99]. `PRIO` qdisc is a classfull-configurable alternative of `pfifo_fast` and enable users to configure the number of queues/bands. In this work, we use a modified version of `PRIO` qdisc. To establish a mapping between the TOS field and queues, we have used `priomap` to associate the TOS values to the bands. The implemented `PRIO` queuing discipline can provide up to $n + 4$ queues in the qdisc layer, where n queues are used

for IoT stations and four queues for the regular stations. Also, the per-queue statistics (number of packets in each queue) is maintained by the `PRIO` qdisc kernel module and communicated to S-KNL module through netlink sockets.

5.5 Simulation Results

This section presents performance evaluation in large-scale scenarios using a simulation tool that we have developed using OMNet++ [103]. The reason we present simulation results before empirical results is that the performance benefits of Wiotap are in particular revealed in scenarios with a large number of IoT stations.

The AP is placed in the center of a 50m×50m area. Both regular and IoT stations are randomly placed in this area. The IoT stations are normally in sleep mode. Each IoT station wakes up every 4 seconds and reports an event to a server. The server, in response, sends back a reply to the station. We refer to this process as a *transaction*. The tail time duration is 10ms and beacon interval is 100ms. Regular traffic intensity refers to the percentage of AP bandwidth utilized by regular stations. When this percentage is between 95% to 99%, we refer to it as near-saturation ($NSat$). To measure energy efficiency, we compute the average duty cycle of all IoT stations.

We compare the performance of the proposed approach against two baselines: (i) R-AP: a regular AP that does not perform any IoT flow prioritization, and (ii) SQ-AP: a single-queue IoT prioritization mechanism that pushes IoT packets in the base queue Q_0. Please note that the second approach employs a simple FIFO scheduling of IoT traffic and does not offer deadline-based intra-IoT traffic prioritization. Each point shows the median of 50 experiments where each experiment includes 30 transactions. Error bars demonstrate upper and lower quartiles.

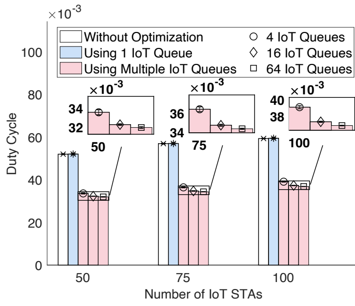

Fig. 5.3: Impact of the number of IoT stations on the average duty cycle of IoT stations. No regular traffic is present on the channel

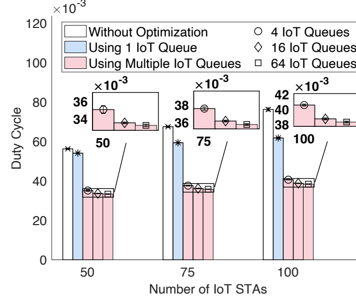

Fig. 5.4: Impact of the number of IoT stations on the average duty cycle of IoT stations. The regular traffic type and intensity are AC_BK and 75%, respectively.

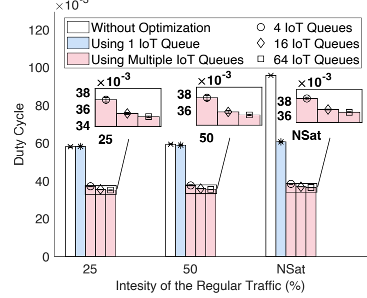

Fig. 5.5: Impact of the amount of regular traffic on the average duty cycle of IoT stations. The number of IoT stations is 75. The regular traffic type is AC_BK.

Figure 5.3 shows the duty cycle per IoT station versus network density. Even when the number of stations is 50, the duty cycle achieved with Wiotap is 37% less than that of R-AP (and SQ-AP). Please note that in this figure, R-AP and SQ-AP show similar results because there is no difference between assigning IoT packets to Q_0 or a regular queue (i.e., Q_{net}^{reg}) when no regular traffic is present. Therefore, these results reveal the main advantage of Wiotap regarding deadline-aware packet scheduling in large-scale IoT networks by using multiple IoT queues and assigning per-packet priority levels among the IoT stations.

Next, we keep the regular traffic constant at 75% and increase the number of IoT stations. Figure 5.4 presents the results. Compared to R-AP and SQ-AP, Wiotap shows 39% and 37% reduction in duty cycle when the number of IoT stations is 50, respectively. Also, when the number of IoT stations is increased from 50 to 100, R-AP and Wiotap show around 35% and 15% increase in duty cycle, respectively. Although SQ-AP shows lower duty cycle compared to R-AP by prioritizing IoT packets over regular traffic, its performance is significantly lower than that of Wiotap since it does

not perform deadline-based scheduling.

Figure 5.5 depicts the performance improvement achieved by Wiotap versus the intensity of regular traffic when 75 IoT stations exist in the network. The average performance improvement in the presence of 25% regular traffic compared to R-AP and SQ-AP are 39% and 38%, respectively. By "average" we refer to the performance improvement of the proposed approach when using 4, 16, and 64 IoT queues compared to the baselines. Also, in the presence of NSat regular traffic, the average performance improvement compared to R-AP and SQ-AP are 61% and 38%, respectively. These results, in particular, show the impact of increasing regular traffic on the energy efficiency of IoT stations when R-AP or SQ-AP are used. For example, when using R-AP, increasing regular traffic from 25% to NSat increases the duty cycle of IoT stations by 65%. Even in high-capacity networks with a few number of regular devices, an attacker might generate a DoS attack to saturate the network and compromise the energy efficiency of IoT stations.

Figure 5.6(a) shows the impact of the number of queues on duty cycle. We observe a decaying trend in duty cycle for all the test conditions because increasing the number of IoT queues enhances the granularity of intra-IoT traffic prioritization. This figure shows that increasing the number of IoT queues from 2 to 4 results in 28% lower duty cycle on average for all the cases. After that, the duty cycle is reduced by around 5% when the number of queues is increased to 64. This behavior is because not all the queues available would be fully utilized as the queue filling rate depends on the amount of concurrent traffic. In general, certain conditions raise the importance of higher-granularity packet scheduling: increasing the number and traffic rate of IoT stations, increasing the standard deviation of RTT, various tail time values of IoT stations. As Figure 5.6(b) shows, increasing the mean and standard deviation of RTT enhances the benefits of using more number of queues.

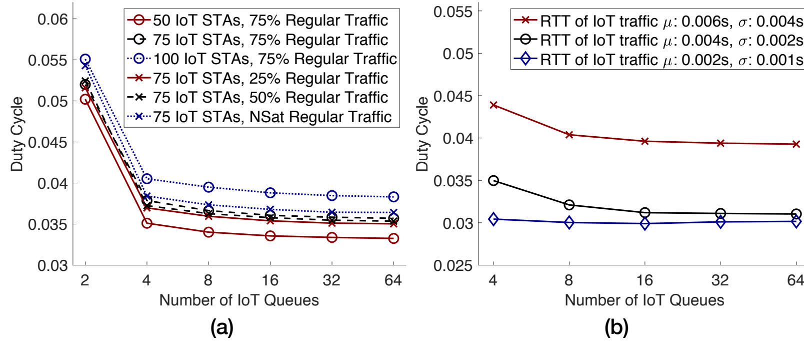

Fig. 5.6: Impact of the number of IoT queues on the duty cycle of IoT stations. (a): Duty cycle versus the number of IoT stations and regular traffic. (b): Duty cycle versus RTT delay in the presence of 100 IoT stations and no regular traffic.

5.6 Empirical Evaluation

This section presents the testbed setup methodology and the empirical performance evaluation of the proposed system. Although Wiotap has been primarily designed to ensure deadline-aware scheduling of large-scale IoT networks, the results of this section show that the proposed solution enhances performance even in networks with a small number of IoT stations.

5.6.1 Testbed

Figure 5.7 shows the testbed architecture. The testbed has been set up as follows.

5.6.1.1 Hardware

We used Cypress CYW43907 [7, 104] as MQTT clients. CYW43907 offers a low-power design, is equipped with an ARM Cortex-R4 processor, and supports 802.11g/n. The AP has been implemented using an Intel NUC machine, which includes an Intel

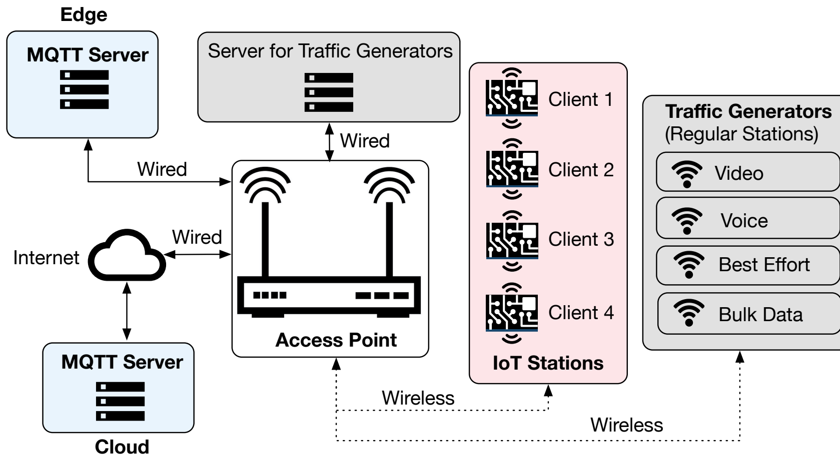

Fig. 5.7: Testbed architecture. Four IoT stations communicate with the AP in the presence of regular traffic. The IoT stations use MQTT to communicate with the broker.

Wireless-AC 7265 card acting as the AP interface. We also added a Linksys AE1200 N300 to the AP for the WiLog module. We configured the AP to operate in 802.11e mode with 54Mbps capacity.

5.6.1.2 Publish/subscribe model

Due to its widespread use in IoT applications, we have adopted MQTT [105], which is based on the publisher/subscriber model. The broker is the point of contact between the publisher and the subscribers, and maintains a list of topics. Each publisher publishes to a topic, and the broker forwards the published messages to all the subscribers interested in the topic. In this chapter, we refer to the process of publishing a message and receiving a response as a *transaction*.

5.6.1.3 Background traffic

Many media-centric applications (such as video calling and gaming) use UDP as the transport layer protocol [106] [107]. In addition, recently, application layer protocols

over UDP, such as Google's Quick UDP Internet Connections (QUIC) protocol is being widely adopted, representing over 7% of all the traffic on the Internet [108, 109]. Hence, UDP traffic can easily saturate a heterogeneous WiFi network by consuming most of the bandwidth [110, 111]. To mimic this behavior, we generate UDP traffic using a C program that runs on a client and sends data flows to the traffic generator server. The program is capable of both setting the class of services (COS) to associate an AC with each packet and controlling inter-packet transmission delays to adjust the amount of channel utilization. A similar program runs on a server to continuously sends back the received UDP packets to the traffic generator.

5.6.1.4 Energy measurement platform

In order to measure energy consumption, we have used EMPIOT [112]. This platform enables us to accurately measure the energy consumption of a code snippet running on the IoT devices. Specifically, by annotating the code running on the IoT boards, the start and stop of energy measurement can be accurately controlled. The EMPIOT platform's basic sampling rate is 500Ksps, which are then averaged and streamed to the control software as 1Ksps. The maximum accuracy error of this platform is 4% compared to the existing high-end commercial products.

The IoT client boards (CYW43907) include components that increase the minimum achievable energy consumption compared to the SoCs only. Therefore, in order to merely take into account the energy consumption of 802.11 communication, we need to collect the base energy consumption of the board when the transceiver is in sleep mode. Our measurements show that the base current consumption is approximately 160mA, and wireless communication increases this value to about 250mA. By subtracting the base power consumption from the results collected during experiments, we report the average energy consumption of the board per MQTT transaction.

5.6.2 Methodology

The testbed has been used to run a series of experiments in the presence of various types and rates of background traffic. To represent a request-response scenario, IoT stations subscribe to their published topic to ensure that the client will receive the published messages from the broker. When referring to energy in the results, we show the energy consumed by publishing a message and getting the reply back, which is referred to as a transaction. We perform the experiments considering MQTT's QoS 1 and QoS 2 modes. The former ensures that the message is delivered at least once, and the latter ensures the message is delivered exactly once. Although QoS 2 has higher overhead in terms of latency and number of packets exchanged while publishing a message, it is the preferred QoS mechanism for mission-critical applications. We run 50 transactions for each AC and background traffic rate, and depict the median and error bars to show the lower and higher quartiles. All the experiments were conducted after 12am to minimize the impact of nearby APs.

Based on the location of the MQTT broker, we have implemented edge and cloud computing scenarios. In the edge scenario, the broker is directly connected to the AP through an Ethernet cable. The mean and standard deviation of RTT are around 3ms and 2ms in this scenario, respectively. In the cloud scenario, we have placed the MQTT broker in a server located in Oregon, US, and the AP and IoT devices are located in Santa Clara, US. We observed that the mean and standard deviation of RTT are 30ms and 10ms, respectively. These scenarios, in particular, enable us to see the impact of round-trip-time (RTT) on energy consumption. Specifically, the RTT of edge and cloud scenarios are less than and more than the tail time, respectively.

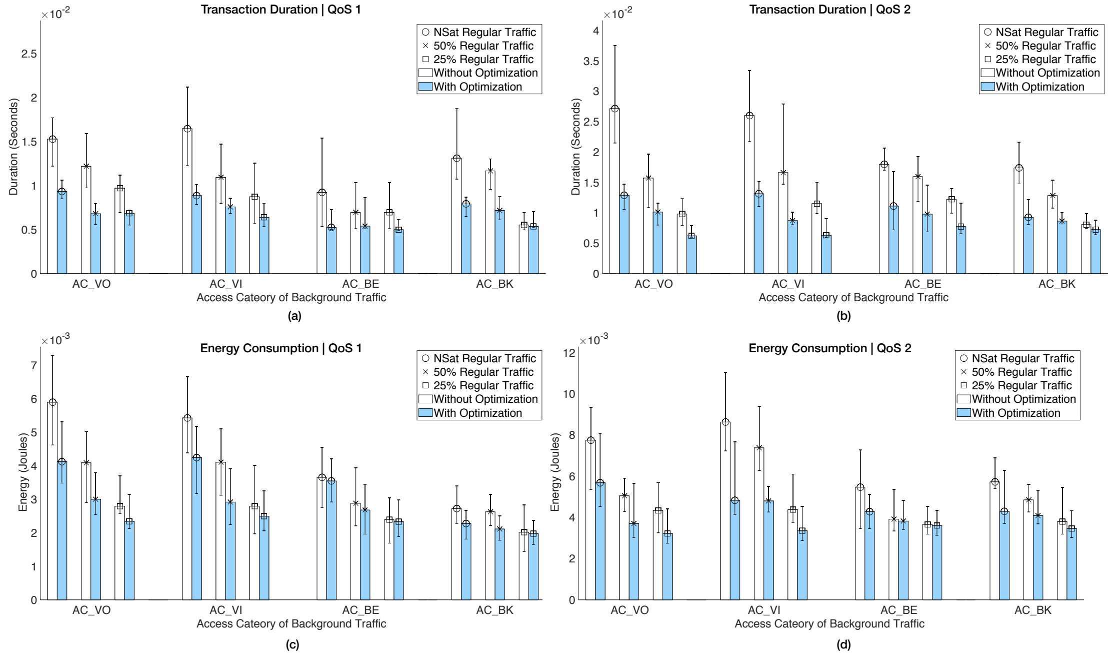

Fig. 5.8: Average delay per MQTT transaction versus the different levels of background traffic in the edge computing scenario. (a) and (b) show transaction duration, and (c) and (d) show transaction energy consumption.

5.6.3 Results and discussions

Figures 5.8 and 5.9 demonstrate the result for the edge and cloud scenarios, respectively. The maximum and average performance improvement of Wiotap in the edge scenario are 52% and 36% in terms of delay and 44% and 18% in terms of energy. For the cloud scenario, the maximum and average performance improvement are 41% and 18% in terms of delay and 38% and 13% in terms of energy.

When the broker is at the network edge, the response packets usually reach the AP within the tail-time of the station. Depending on the deadline of this packet compared to other IoT stations, Wiotap prioritizes the packet to ensure its delivery before its deadline. In the cloud computing scenario, the network latency is usually larger than the tail-time, and stations will have to habitually wake up again during the next beacon interval to retrieve downlink packets from the AP. But this time, as our

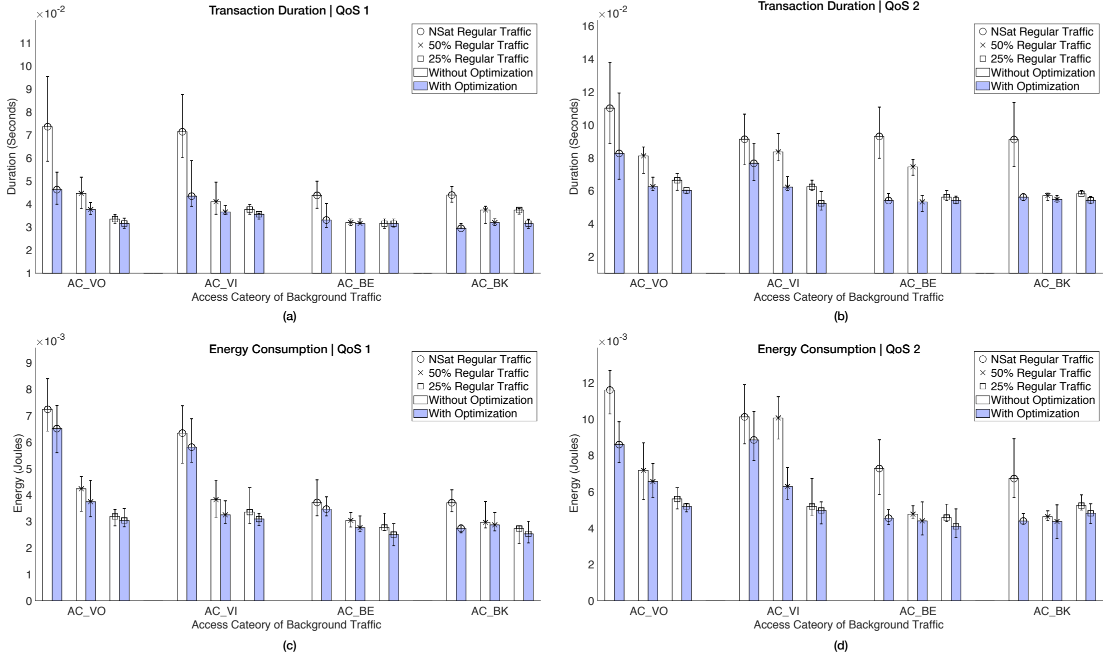

Fig. 5.9: Average delay per MQTT transaction versus the different levels of background traffic in the cloud computing scenario. (a) and (b) show transaction duration, and (c) and (d) show transaction energy consumption.

solution prioritizes the packets over background traffic, the station spends less time in idle listening mode. Therefore, compared to the edge scenario, more energy is spent in the cloud scenario on average per station per transaction.

As the results show, the energy consumption of IoT station is higher in the presence of AC_VO compared to AC_BK and AC_BE. Besides, the rate of background traffic has higher impact on the energy consumption of IoT station for higher-priority background flows. We justify this behaviour as follows. The regular traffic regularly fills up the EDCA queues of corresponding ACs. Whenever a burst of IoT traffic occurs, the priority of IoT packets is promoted at the qdisc layer. However, since we do not modify the 802.11e EDCAF (Enhanced Distributed Channel Access Function) at the MAC layer, although the IoT traffic has the highest priority, it would only have a higher chance of being transmitted while contending with the existing traffic in the EDCA queues. In

a statistical sense, the transmission probability of IoT traffic is higher compared to lower priority ACs due to the lower values of CW_{min}, CW_{max}, and $AIFS$ for high-priority ACs (IoT traffic). However, the random backoff time chosen by lower-priority ACs adds some amount of uncertainty, resulting in scenarios, such that lower-priority flows gain channel access before higher-priority flows [113]. More specifically, to avoid the starvation of low-priority flows, the 802.11e MAC layer prioritization only offers a higher probability of transmission for higher priority ACs, and does not guarantee that the higher priority AC packets will always be sent before the lower priority packets [114, 115].

5.7 Summary

In this chapter, we proposed Wiotap, which is an 802.11 access point serving IoT and regular stations. Wiotap enhances the energy efficiency and timeliness of IoT stations in large-scale networks by applying per-packet scheduling of IoT packets based on their power state. In addition, Wiotap ensures the high efficiency of IoT stations in the presence of regular traffic and protects them against DoS attacks.

The performance and applicability of the proposed approach can be enhanced by integrating context awareness. For example, detecting application layer protocol eliminates the burden of manually identifying IoT devices. Also, analyzing per-device traffic pattern enables the AP to control the power status of stations by sending management packets. To extend the proposed mechanism to scenarios with multiple APs and mobile stations, a software-defined networking architecture can be employed to collect the required data (such as packet laxities) from multiple APs and run the proposed algorithms centrally.

This page is intentionally left blank

Chapter 6

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

6.1 Introduction

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. Adjusting the sleep schedule or the STAs, such the STAs transition to the low-power sleep until the DL response is ready to be delivered requires an accurate estimation of the delay between UL and DL 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 chapter, 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. 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.

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 chapter is organized as follows. We present delay components and implementation details of the AP in Section 6.2. We present the edge-assisted sleep scheduling mechanism in Section 6.3. Section 6.4 presents empirical performance evaluations. Section 6.6 summarizes the key contributions of this chapter.

6.2 Delay Analysis and AP Development

6.2.1 Delay Components

As Figure 6.1 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

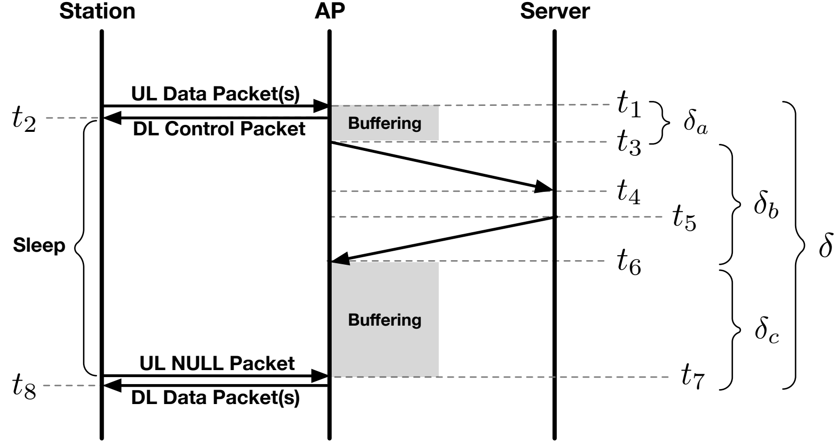

Fig. 6.1: The end-to-end delay components between a station and a server. The prediction of δ_c is particularly challenging because it is affected by several factors such as traffic rate, channel utilization, and buffering mechanisms employed by Linux's qdisc and wireless NIC's driver.

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. The goal of this chapter 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 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.

6.2.2 AP Development

The current AP architectures do not provide the necessary tools to collect and apply predictive scheduling [27]. In this section, we present an AP architecture that allows us to collect the features necessary for predictive scheduling. Figure 6.2 presents

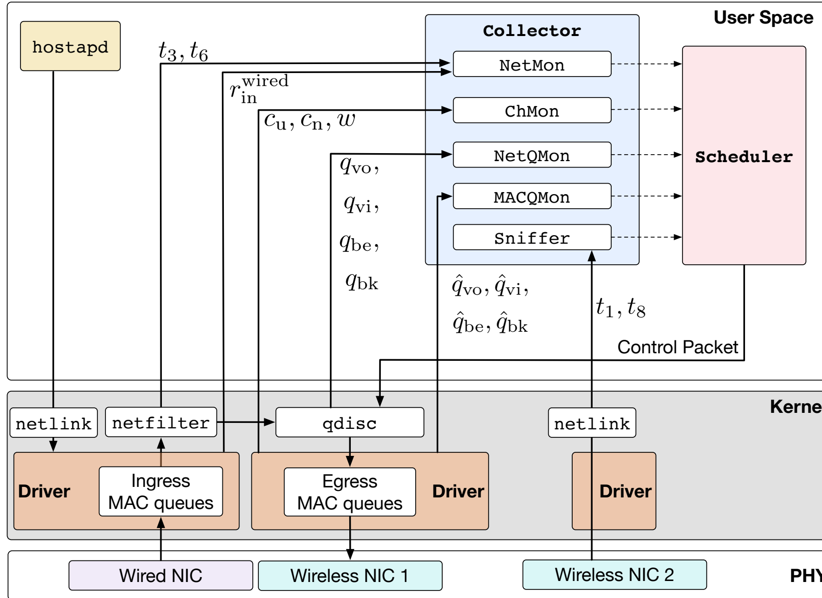

Fig. 6.2: The AP architecture developed and used in this chapter. The Collector module communicates with various kernel and user-space components to collect a set of features required for delay prediction. The `Scheduler` estimates the sleep duration and conveys it to the station. This figure primarily 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.

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` [86], 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.

6.2.3 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 chapter 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 [116]. 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 6.2). 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 [117]. 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 [118, 119]. To this end, we modify the `netfilter` [98] 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.

6.2.4 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 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 [12, 92], 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.

6.2.4.1 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.

6.2.4.2 qdisc queues

Every network interface is assigned a qdisc, which is pfifo_fast by default [90]. 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).

6.2.4.3 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%.

6.2.4.4 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 ToS field. These queues are denoted by $\hat{\mathbf{Q}} = \{\hat{q}_{\mathrm{vo}}, \hat{q}_{\mathrm{vi}}, \hat{q}_{\mathrm{be}}, \hat{q}_{\mathrm{bk}}\}$. 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.

6.2.4.5 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:

$$\begin{aligned} \mathbf{C_u}, \mathbf{C_n}, \mathbf{R}_{\text{in}}^{\text{wired}}, \mathbf{W}, \mathbf{Q}_{\text{vo}}, \mathbf{Q}_{\text{vi}}, \mathbf{Q}_{\text{be}}, \mathbf{Q}_{\text{bk}}, \\ \widehat{\mathbf{Q}}_{\text{vo}}, \widehat{\mathbf{Q}}_{\text{vi}}, \widehat{\mathbf{Q}}_{\text{be}}, \widehat{\mathbf{Q}}_{\text{bk}} \end{aligned} \tag{6.1}$$

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:

$$\begin{aligned} \mathbf{C}_{\text{u}} = [c_{\text{u}}(t' - k \times \Delta), c_{\text{u}}(t' - (k-1) \times \Delta), ..., \\ c_{\text{u}}(t' - \Delta), c_{\text{u}}(t')] \end{aligned} \tag{6.2}$$

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.

6.2.5 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 6.1), 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.

6.3 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.

6.3.1 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 [120]. Resembling 802.11 traffic, Figure 6.3 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}.

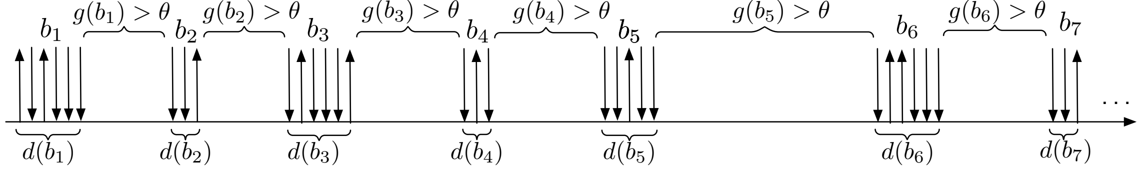

Fig. 6.3: Traffic characterisation.

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 [120], capturing network dynamics can be achieved by focusing on characterizing burstiness. Additionally, Xiao et al. [121] 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 char-

acterized 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:

$$\mathcal{B} = \left(1 - \frac{1}{\mathcal{M}}\right) \times \left(\frac{\sum_{i=1}^{N} s(b_i)}{N}\right) \tag{6.3}$$

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:

$$\begin{aligned} \mathcal{D} = \frac{1}{N}\sum_{i=2}^{N} \frac{\frac{|d(b_i)-d(b_{i-1})|}{d(b_{i-1})}}{g(b_{i-1})} + \frac{1}{N}\sum_{i=2}^{N} \frac{\frac{|s(b_i)-s(b_{i-1})|}{s(b_{i-1})}}{g(b_{i-1})} + \\ \frac{1}{N}\sum_{i=2}^{N} \frac{\frac{|p(b_i)-p(b_{i-1})|}{p(b_{i-1})}}{g(b_{i-1})} + \frac{1}{N}\sum_{i=2}^{N} \frac{z(b_i)}{g(b_{i-1})} \end{aligned} \tag{6.4}$$

where,

$$\begin{aligned} z(b_i) = \frac{|p_{\text{vo}}(b_i) - p_{\text{vo}}(b_{i-1})|}{p_{\text{vo}}(b_{i-1})} + \frac{|p_{\text{vi}}(b_i) - p_{\text{vi}}(b_{i-1})|}{p_{\text{vi}}(b_{i-1})} + \\ \frac{|p_{\text{be}}(b_i) - p_{\text{be}}(b_{i-1})|}{p_{\text{be}}(b_{i-1})} + \frac{|p_{\text{bk}}(b_i) - p_{\text{bk}}(b_{i-1})|}{p_{\text{bk}}(b_{i-1})} \end{aligned} \tag{6.5}$$

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.

6.3.2 Data Collection

We use the metrics mentioned above and compare datasets generated in our testbed against those collected in multiple real-world environments. Figure 6.4 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 §6.2, 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.

6.3.3 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 sam-

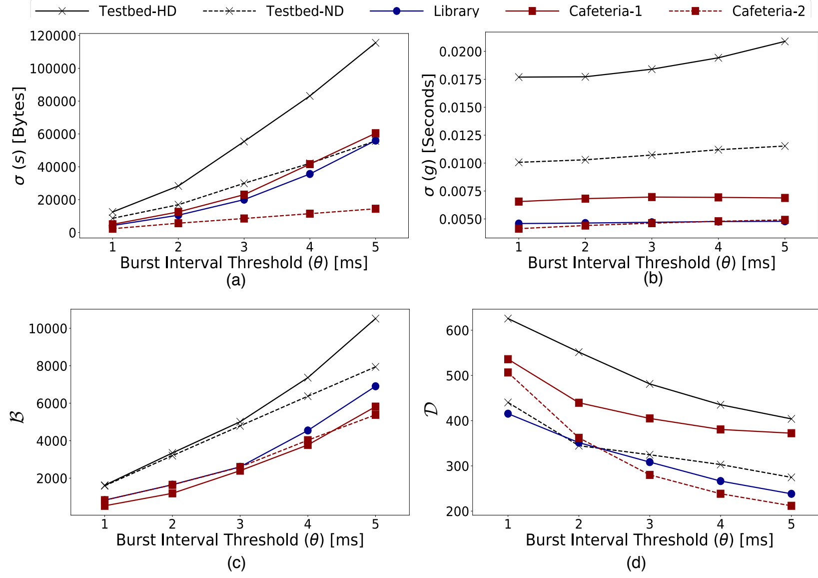

Fig. 6.4: (a) Standard deviation of burst size, (b) standard deviation of burst interval, (c) burstiness ($\mathcal{B}$), and (d) dynamicity ($\mathcal{D}$) of traffic generated by our testbed compared to traffic captured in real-world environments. ND and HD refer to normal and high dynamicity, respectively.

ples (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.

6.3.4 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. 6.2, 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 $c_{\mathrm{u}}(t')$ and $c_{\mathrm{u}}(\bar{t}) = \sum_{i=1}^{k} w_i \times c_{\mathrm{u}}(t' - i \times \Delta)$ for channel utilization. Here, $c_{\mathrm{u}}(\bar{t})$ 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 §6.2.4.5, 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 LSTM [122], 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 [123]. 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 [124]. 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

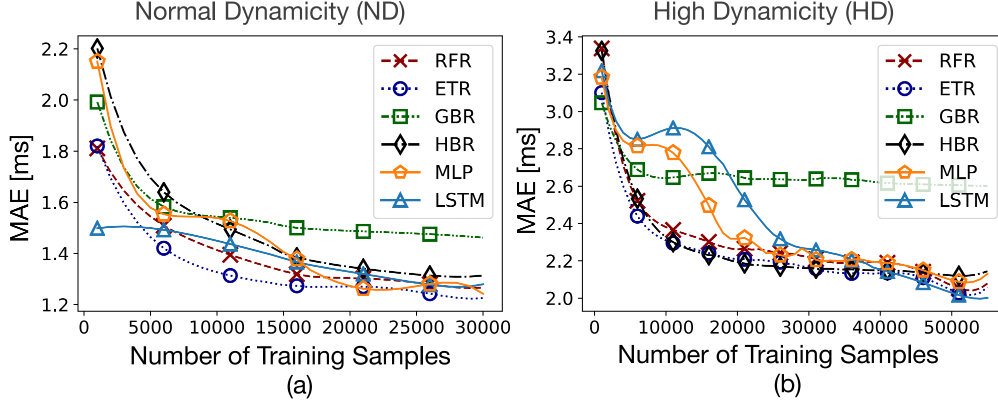

Fig. 6.5: MAE of machine learning algorithms versus the number of samples (transactions) in training dataset for (a) Normal Dynamicity (ND), and (b) High Dynamicity (HD) scenarios. Results are averaged over all ACs. ETR converges the fastest, and LSTM requires up to 3x more data points compared to ETR.

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 [125] with learning rate of 0.01.

6.3.5 Model Evaluation

We used the test dataset for all evaluations. Figure 6.5 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 6.5. 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

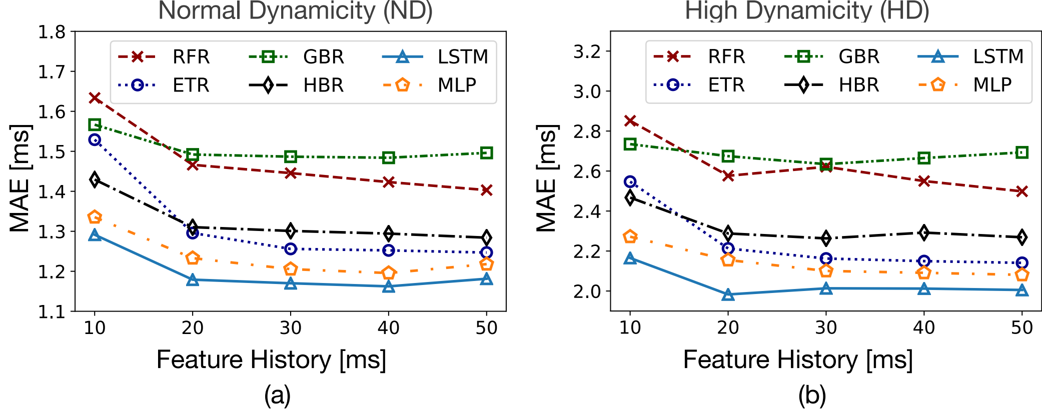

Fig. 6.6: MAE of machine learning algorithms with respect to feature history in (a) Normal Dynamicity (ND), and (b) High Dynamicity (HD) scenarios. Results are averaged over all ACs.

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 chapter, we use the required number of data points that are needed by each algorithm for performance convergence.

Figure 6.6 quantifies the effect of feature history (k in Eq. 6.2). 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, 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 6.7 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

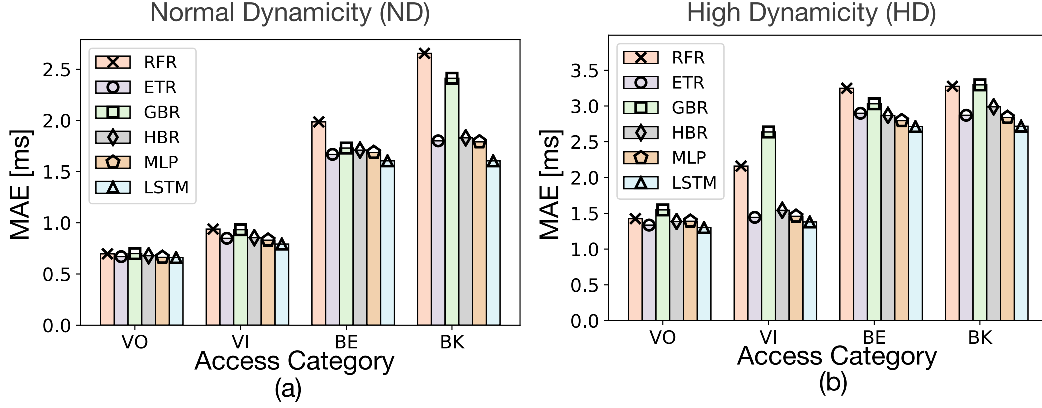

Fig. 6.7: MAE of machine learning algorithms versus transaction's AC in (a) Normal Dynamicity (ND), and (b) High Dynamicity (HD) scenarios. MAE of VO and VI packets is lower than BK and BE packets.

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 6.7 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 6.8 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 ±5.3 ms in case of ND scenario and ±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.,

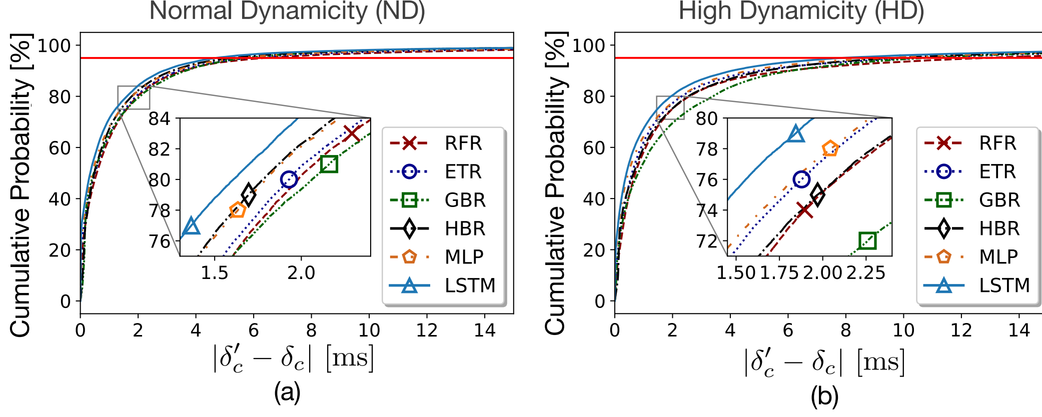

Fig. 6.8: ECDF of prediction errors ($|\delta'_c - \delta_c|$) while utilizing various machine learning algorithms in (a) Normal Dynamicity (ND), and (b) High Dynamicity (HD) scenarios. All machine learning algorithms are able to predict δ'_c for 95% of the packets with an error of ±5.3 ms in case of ND scenario, and ±10.6 ms for the HD scenario. We have used markers in the inset graph for better visibility.

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 6.9 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 6.10 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

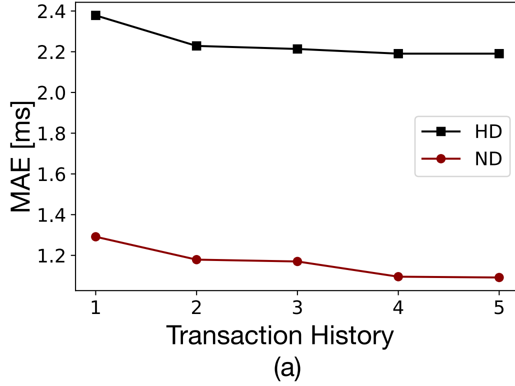

Fig. 6.9: Effect of transaction history on MAE of LSTM.

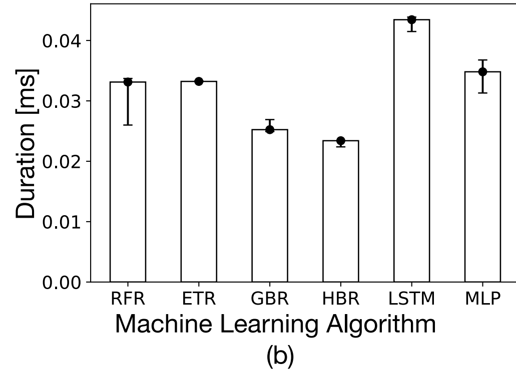

Fig. 6.10: Processing time of the prediction algorithms.

duration is 48μs/207μs.

6.4 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 (§6.3.5) 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 6.5). Hence, in scenarios where it is not possible to collect large datasets for training, either ETR or MLP can be used.

6.4.1 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 [7], 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.[1] 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 [112] 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 6.1): *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.

6.4.2 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

[1] Note that the case where multiple uplink and downlink packets are exchanged is simply supported as explained in §6.2.

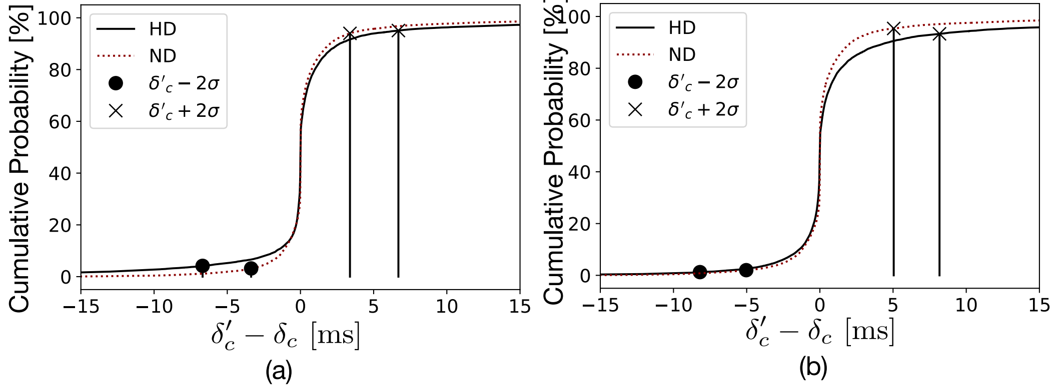

Fig. 6.11: Cumulative distribution function of prediction error ($\delta'_c - \delta_c$) for (a) voice, and (b) background ACs. Prediction error of voice AC is lower than that of background AC. Depending on the application's energy-delay tradeoff, the station may wake up before, on, or after the predicted time.

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 6.11 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 $\delta' - 2\sigma > 0$.

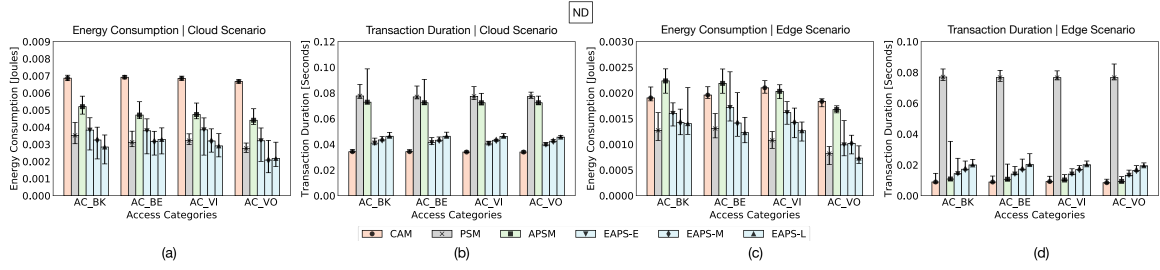

Fig. 6.12: Performance comparison of EAPS with 802.11 power saving mechanisms in *ND* conditions for all ACs. (a) and (b) show the average *per-transaction* energy and duration in cloud scenario, respectively. (c) and (d) show the average *per-transaction* energy and duration in edge scenario, respectively.

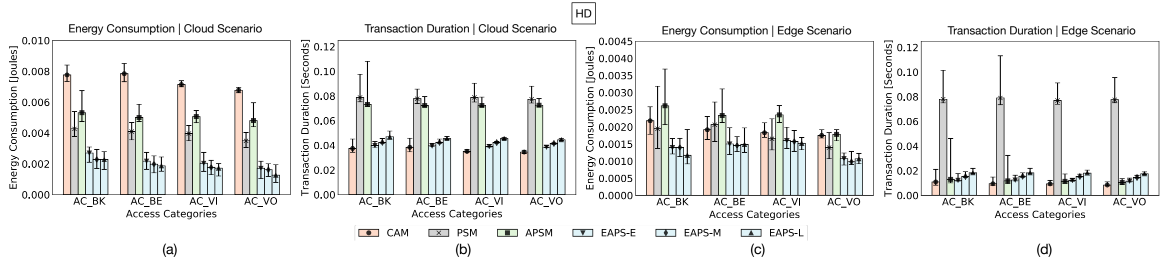

Fig. 6.13: Performance comparison of EAPS with 802.11 power saving mechanisms in *HD* conditions for all ACs. (a) and (b) show the average *per-transaction* energy and duration in cloud scenario, respectively. (c) and (d) show the average *per-transaction* energy and duration in edge scenario, respectively.

6.4.3 Results

Figures 6.12 and 6.13 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 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.

6.5 Discussion

Wake-up radio. WUR mechanisms such as 802.11ba [46, 47] 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 [46], 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 §6.2.3, the primary types of networks used in this chapter 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 §6.2.3 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.

6.6 Summary

In this chapter, we proposed 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.

Chapter 7

Traffic Characterization for Efficient TWT Scheduling in 802.11ax IoT Networks

7.1 Introduction

The newly-introduced 802.11ax standard provides a method called TWT for assigning *service periods* to STAs. Compared to the earlier power-saving modes, TWT allows for potentially higher energy efficiency and throughput. Specifically, by properly assigning service periods to STAs, channel contention reduces, packet buffering delay in the AP drops, and packet aggregation efficiency enhances [33]. Nevertheless, to realize the benefits of TWT scheduling, accurate characterization of the traffic flows of STAs is required by an AP to allocate service periods that address STAs' traffic requirements [34, 33, 35].

In this chapter, we argue that accurate characterization of the traffic pattern of STAs is necessary to allocate TWT service periods that result in both high throughput and energy efficiency. For example, consider an IoT device collecting a batch of sensor samples every few seconds. The process of sample collection from an Analog-to-Digital Converter (ADC), packet preparation, and transfer from the application subsystem to the NIC introduce a non-negligible inter-packet interval that would result in bandwidth waste if not utilized by other STAs. In this chapter, we study and reveal that the ex-

isting traffic characterization methods demonstrate the following shortcomings: (i) CU provides only the accumulated channel time usage by all the STAs; (ii) packet sniffing approach is affected by channel access delay and collision; (iii) in addition to being affected by channel access delay, some devices generate BSR only when requested by the AP, and also, the reported values are fixed for all the MPDUs inside an A-MPDU. To address these shortcomings, we propose Source-assisted Traffic Characterization (SATRAC), a method similar to In-band Network Telemetry (INT) in the traffic generation source. We leverage the eBPF technology to add the difference between packet generation time instances to the TCP Options field (for TCP traffic) or IP Options (for UDP traffic). This information allows the AP to accurately and quickly determine traffic generation patterns and assign TWT service periods to STAs. We compare the performance of SATRAC against the existing methods and show its superior performance and robustness against factors such as CU and channel access delay.

The rest of this chapter is organized as follows. Motivation for traffic characterization is given in Section 7.2. Section 7.3 studies the shortcomings of existing methods. We present the design and evaluation of SATRAC in Section 7.4. We summarize a few salient features of this chapter in Section 7.5.

7.2 Traffic Pattern Analysis

This section analyzes the characteristics of network traffic generated in real-world IoT scenarios. We separate micro-bursts and macro-bursts and justify the importance of traffic characterization for TWT assignment.

We consider the following IoT scenarios: (i) **Sensor**: We use an RTOS development kit (CYW54907) for collecting accelerometer readings. Whenever the device enters the sample collection phase, it collects 3920 samples, which results in 5880 bytes

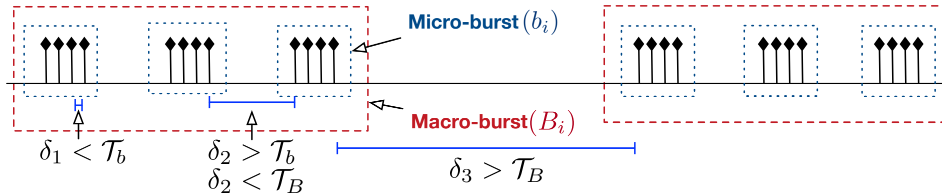

Fig. 7.1: Micro-burst and macro-burst characterization.

((3920 × 12 bits)/8); these bytes are then sent via a TCP connection. (ii) **Camera**: We built a security camera using Raspberry Pi (RPi) camera module (version 2) attached to a RPi 3B+. The camera continuously captures and sends images via a TCP connection. Each images is processed by the H.264 codec. (iii) **Video Streaming**: A YouTube video is streaming on an Amazon Echo Show device. All the experiments were run in interference-free environments.

To build a generalized traffic analysis framework, we consider three inter-packet intervals and use the traffic structure of Figure 7.1. A *micro-burst*, denoted as b_i, is defined as a train of packets with inter-arrival time less than a specific threshold value $\mathcal{T}_b$ [120, 50]. The interval between packets in a micro-burst is denoted as δ_1. If the interval between two packets is larger than $\mathcal{T}_b$ but less than $\mathcal{T}_B$, a new micro-burst is detected. The interval between micro-bursts is denoted as δ_2. If the interval between packets is larger than $\mathcal{T}_B$, a new macro-burst is detected. A macro-burst is represented as B_i, and the interval between macro-bursts is denoted as δ_3.

Figure 7.2 presents the results for the Sensor scenario.

We observe that even within a micro-burst, the mean interval between packets (δ_1) is about 400 μs. This delay is caused by packet preparation delay and the timing requirements of the 802.11 standard (e.g., channel access contention, SIFS, DIFS). Regarding packet preparation delay, we modified the code and added probes to each stage of the packet preparation process and observed that, for example, the transmission of a packet from driver to NIC introduces a non-negligible delay of about 28 μs. Comparing

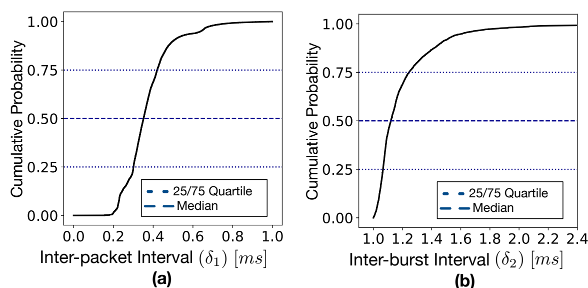

Fig. 7.2: Inter-packet intervals for the Sensor scenario. δ_1 is caused by packet preparation delay and transmission parameters of 802.11 standard. The difference between δ_1 and δ_2 is due to sample collection delay.

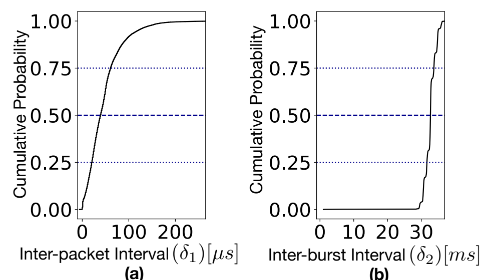

Fig. 7.3: Inter-packet intervals for the Camera scenario. δ_2 is about 33 ms, corresponding to 30 fps.

Figures 7.2(a) and (b), the interval between micro-bursts is affected by sample collection delay. Specifically, this delay is caused by the communication between the processor and ADC over the Serial Peripheral Interface (SPI) to collect samples [126]. Therefore, δ_2 would increase if a higher number of samples must be collected per round.

Figure 7.3 shows the inter-burst intervals for the Camera scenario. Each micro-burst constitutes multiple packets. The camera captures a frame and then prepares multiple packets to send the frame. The amount of data in each frame depends on the resolution of the video stream requested (e.g., 480p, 720p, 1080p). As we see in Figure 7.3(b), the interval between micro-bursts (δ_2) is about 33 ms, which corresponds to 30 fps. Figure 7.4 shows the results for video streaming scenario. The mean interval between macro-bursts (δ_3) is 10 seconds, the mean interval between micro-bursts (δ_2) is 2 ms, and the mean interval between packets of a micro-burst (δ_1) is 9 μs.

These studies demonstrate the intervals between packets in a micro-burst, the intervals between micro-bursts, and the interval between macro-bursts. Characterizing these delays is essential for three purposes: (i) allocating TWT service periods based on each STA's demands, (ii) utilizing inter-packet intervals by other STAs to enhance

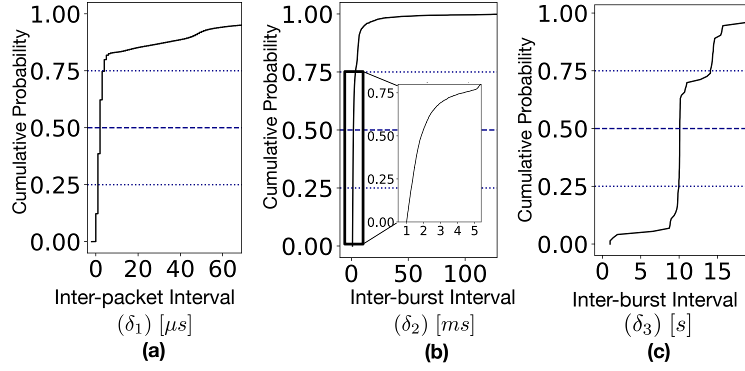

Fig. 7.4: Inter-packet intervals for the Video Streaming scenario.

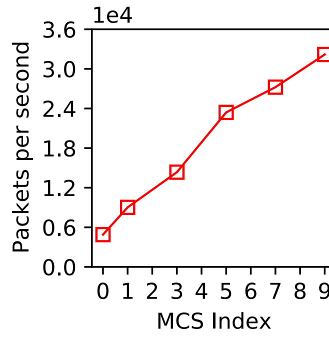

Fig. 7.5: Packets per second with respect different MCS values

throughput, and (iii) enhancing packet aggregation performance, which results in shorter communication delays and higher energy efficiency.

For example, suppose a TWT service period ends before sending all the packets of a micro-burst. In this case, the STA either needs to wait until the next TWT service period (causing communication delay) or contend with other STAs for channel access (lower energy efficiency).

To confirm the feasibility of utilizing inter-packet intervals within a micro-burst by other STAs, we measure the actual packet (1500 bytes) transmission rate in an 802.11ax testbed.

As Figure 7.5 shows, with MCS 5, a STA can send about 24000 packets/second, which means the duration of each packet is around 41 μs. Therefore, in most of the

aforementioned cases, one or more STAs can utilize inter-packet and inter-burst intervals.

7.3 Traffic Characterization via Channel Utilization, BSR, and Packet Sniffing

In this section, we study the shortcomings of the three available and most-widely used traffic characterization methods.

7.3.1 Channel Utilization (CU)

CU is defined as $t_{activity}/t_{total}$, where $t_{activity}$ is the time duration the NIC has sensed signal power higher than a pre-specified threshold value during time duration $t_{overall}$. CU values can be collected from the driver via various methods such as the 'proc' file system (`procfs`) in Linux. However, since the information provided by CU is cumulative, it cannot be used to characterize per-STA traffic patterns.

7.3.2 Packet Sniffing

Several real-world deployments and COTS enterprise APs utilize an external NIC operating as a sniffer to monitor the traffic patterns of STAs [36]. Since each AP acts as the central point of communication for all the traffic to and from the STAs, collecting an AP's driver logs can also be utilized for determining the traffic pattern of STAs.

The shortcomings of this approach are as follows. First, in addition to requiring extra hardware (e.g., additional NIC), the sniffed packets must be processed by the AP's

operating system and user application; thereby increasing processing overhead. Second, any inconsistency between the hardware and antenna configurations of the AP's primary NIC and those of the sniffer results in a mismatch between the sniffed packets and those exchanged by the AP's primary radio [127]. Third, and even more importantly, the timestamps of sniffed packets do not represent the actual packet generation instances by STAs. This is due to factors such as channel access contention delay, internal prioritization of packets belonging to different ACs, and packet preparation delay. We will further analyze this method in Section 7.4.

7.3.3 Buffer Status Report (BSR)

In the 802.11ac standard, the Queue Size (QS) sub-field, contained inside the QoS Control field, reports the total data queued in the STA's queues. The AP primarily utilizes this information to allocate TXOP to each STA. BSR is a new functionality recently introduced in the 802.11ax standard to enhance the exchange of information on the transmission buffer size of STAs. For example, compared to the QS field, BSR provides more specific information, such as the Queue size of the highest-priority AC. The Queue Size All (QSA) field of BSR conveys the cumulative amount of data in all queues.

In this section, we reveal the challenges of using BSR for traffic characterization. First, the 802.11ax standard does not mandate the inclusion of queue statistics in each packet. To verify this, we have selected several COTS 802.11ax NICs and noticed that Intel AX200 and Realtek RTL8852A transmit BSR intermittently, based on the amount of traffic queued. In contrast, Compex WLT639 includes a BSR in every packet. Additionally, none of the APs and STAs we evaluated support requesting or generating BSR manually.

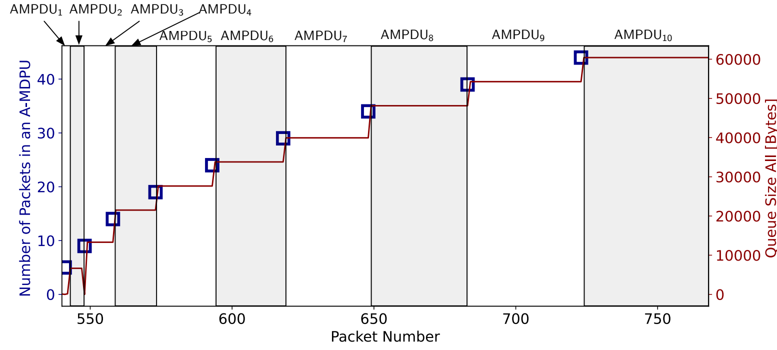

Fig. 7.6: Traffic characteristics and BSR values for a flow with increasing burst size. The blue squares denote the number of packets in the succeeding A-MPDU, and the red curve represents BSR's QSA field of packets received by the AP. The results show that the reported buffer size is fixed for all the MPDUs inside an A-MPDU.

Secondly, by empirical analysis of packet exchange traces, we observed that for those 802.11ax devices that include BSR in each packet, all the MPDUs included in an A-MPDU report the same value even though the payloads they are carrying have been generated at different time instances. The reported value is the state of queues before the transmission of A-MPDU. To demonstrate this behavior, we captured BSR values by a STA using Compex WLT639. Also, to generate various A-MPDU sizes, we gradually increase the amount of data pushed by the application to the transport layer socket. Figure 7.6 shows the number of packets per A-MPDU (left y-axis) and the QSA values in BSR (right y-axis). The squares denote the number of MPDUs in the succeeding A-MPDU, and red curve denotes QSA value reported by each incoming packet. As the results show, the BSR value reported per A-MPDU is fixed and reports the amount of data in the driver's buffer plus the size of A-MPDU being sent. In general, assume the QSA (denoted as Q) values are received at time instances t_n and t_{n+1} from a STA. The amount of traffic generation during this interval can be represented as: $Q_{t_{n+1}} - Q_{t_n} + \sum_{[t_n,t_{n+1})} p_{tx}$, where $Q_{t_{n+1}}$ and Q_{t_n} represent the received BSR values at time t_{n+1} and t_n, respectively, and $\sum_{[t_n,t_{n+1})} p_{tx}$ is the sum of the size of packets transmitted by the STA during time interval $[t_n, t_{n+1})$, which excludes the packet received at

t_{n+1}. In summary, the BSR approach neither provides the actual data generation time instances nor provides real-time snapshots of driver's queue sizes.

Since BSR does not provide any timing information, we augment it as follows to estimate packet generation time instances. We refer to this method as BSR′ and evaluate it in Section 7.4.3. Assume the BSR values received from two packets at time instances t_n and t_{n+1} are Q_{t_n} and $Q_{t_{n+1}}$. If $Q_{t_{n+1}} > Q_{t_n}$, we estimate inter-packet generation instances during interval t_{n+1} to t_n as $(t_{n+1} - t_n)/((Q_{t_{n+1}} - Q_{t_n})/p_{tx})$, where p_{tx} is the packet size. If $Q_{t_{n+1}} < Q_{t_n}$, we use the time instance of sniffed packets.

7.4 Source-assisted Traffic Characterization

In this section, we propose a method named Source-assisted Traffic Characterization (SATRAC) and evaluate it in terms of traffic characterization accuracy and its effect on TWT allocation.

7.4.1 Design and Implementation

The basic idea of SATRAC is that, if we keep track of packet generation time instances in each STA, the AP can construct the traffic pattern of the STA, regardless of the effect of packet preparation delay, channel access contention, interference, and packet loss. To this end, we require each STA to modify packets in their protocol stack's data-path and add timing information—an approach similar to INT. In order to reduce packet overhead, instead of including an absolute timestamp in each packet, we include only a 2-byte value encoding the difference between the generation time of the current packet and the previous packet of the same flow. To this end, the STA computes a unique 5-tuple hash value for each flow and keeps track of the timestamp of the last generated

packet. In this chapter, to simplify compatibility with existing implementations, we chose the TCP header's Options field to include timing information. Alternatively, for non-TCP traffic, the timing information can be added to the IP header's Options field.

In order to add timing information to packets, we consider two approaches, as follows.

7.4.1.1 Packet Modification in the MAC or NIC

As explained in Section 7.2, packet preparation increases inter-packet delay. To capture packet processing delay, we need to add timing information to each packet when it is ready to be sent; therefore, we need to add the timing information when the packet arrives in the NIC. However, since modifying NIC's firmware is infeasible, the alternative is to add timing information when MAC processing completes and add driver-to-NIC handoff delay as a constant value to this delay. The challenge with this approach is that any modification to the TCP header requires recalculation of TCP checksum and MAC checksum, and any changes to the IP header requires MAC checksum recalculation; in both cases, packet preparation overhead is unnecessarily increased.

7.4.1.2 Packet Modification in the TCP Layer

To eliminate the need for checksum recalculation, we add timing information in the TCP layer when the TCP protocol prepares the TCP header. To account for packet preparation delay, we add the delays caused by the IP layer, MAC layer, and driver-to-NIC handoff to the timing information.

A straightforward approach to modifying the TCP Options field on Linux systems is to use `setsockopt`; however, only a specific set of options can be modified with this API. An alternative is to craft a raw packet with appropriate TCP Options fields

hardcoded; nevertheless, this method requires modifying the applications. Instead of these two approaches, we leverage eBPF and build an application-agnostic middleware for setting the TCP Options field (note that this approach can be used for setting the IP Options field as well). This eBPF module can easily be executed from the user-space on each STA to embed the timing information without any modifications to the kernel's code-base. Also, this approach eliminates the need to modify applications, as it acts as a shim layer between applications and the transport layer. eBPF enables real-time patching of the Linux kernel by allowing users to insert user-defined logic (programs) into the kernel. eBPF programs are associated with hook points that are triggered on the execution of either a *syscall* or a kernel function. We use `BPF_PROG_TYPE_SOCK_OPS` program type that allows the modification of socket options on a per-packet basis. When an event, such as `sendmsg` call, TCP connection, or TCP retransmit timeout occurs, `bpf_sock_ops` structure is returned, which provides the context of the event along with the "op" field identifying the source of the event. We hook our eBPF program to the `tcp_write_options` function, which is responsible for adding the TCP Options field. For measuring packet preparation delay, similarly, we use eBPF hooks in the TCP and MAC layer. For driver-to-NIC delay, the driver is modified to measure the duration of Direct Memory Access (DMA) transactions.

Since APs usually run Linux, a similar eBPF program extracts and parses the values included in TCP Options field of packets received from STAs to characterize uplink traffic. For characterizing downlink traffic, the same AP module keeps track of the packet arrival instances on the AP's wired NIC for each STA. In this chapter, we primarily address characterizing uplink traffic though.

The implementation of SATRAC on RTOS-based STAs depends on their protocol stack used. On a platform using ThreadX with NetXDuo stack, we simply added timers to the TCP and driver codes to measure packet preparation delay. Also, to

embed the timing information in TCP Options field, we modified the TCP code.

7.4.2 Analytical Comparison

Figure 7.7 presents a simple scenario. Row (a) shows that at each time instance t_1, t_2, t_3 and t_4, the application generates ψ bytes. The interval between data generation instances is denoted as Δ. Row (b) shows that each ψ-byte message is segmented into three packets. The interval between these packet generations, denoted as δ_1, depends on packet preparation delay. The third (c) and fourth rows (d) present the actual packet transmission instances (note that the time instances are similar).

When using BSR (row (c)), the STA includes in each packet the amount of data in the buffer. For example, the packet generated at time t_1 is transmitted at time $t_{2,1}$, and this packet indicates only the buffer size at the beginning of packet transmission (2ψ). Since the packet does not convey packet generation time, this data could have been generated any time during the time interval between the transmission of this packet and the previous packet. Additionally, if the four packets transmitted sequentially at time instance $t_{3,1}$ are aggregated as an A-MPDU, all these four packets report value 2ψ (as explained in Section 7.3.3), further affecting the accuracy of traffic characterization. Similarly, the packet sniffing method cannot be used for identifying packet generation times. Specifically, as it can be observed, the packet transmission instances do not represent packet generation instances. Also, note that BSR and packet sniffing methods cannot be used to determine a STA's required CU during a specific time period because we need to know the interval during which packets have been generated to calculate CU.

Using MonFi, the difference between packet generation timestamps is added to each packet. Regardless of packet transmission time, the first packet of each micro-burst

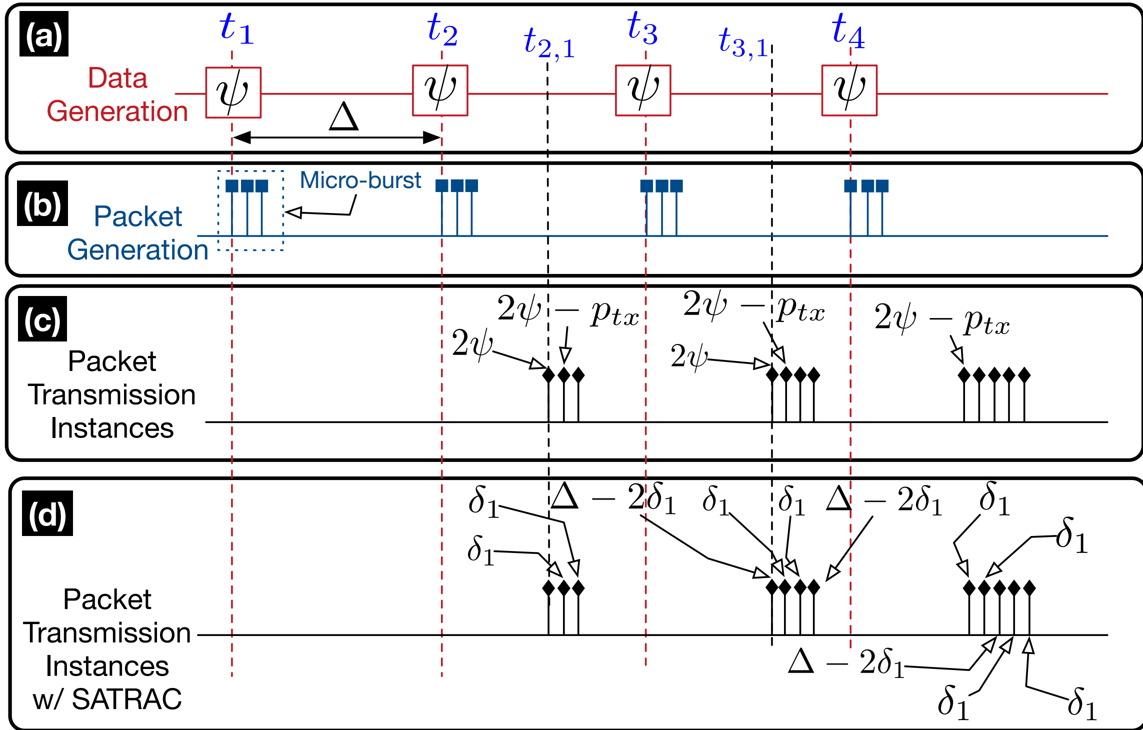

Fig. 7.7: (a): Data generation time instances by application. (b): Packet generation time instances. (c): Packet transmission instances. This row shows the time instance of packet capture by sniffer, as well as the BSR value of each packet. (d): Packet transmission instances and timing information added to each packet by SATRAC.

has the time stamp $\Delta - 2\delta_1$, and the second and third packets include timestamp δ_1. This method allows the AP to determine both Δ and δ_1 to characterize traffic accurately.

7.4.3 Empirical Evaluations of SATRAC

In this section, we empirically compare the performance of SATRAC versus packet sniffing (Section 7.3.2) and BSR′ (Section 7.3.3). We use an 802.11ax testbed including one AP and multiple STAs running Linux. We characterize the accuracy of traffic characterization for one STA. Other STAs are used to introduce variations in CU. We consider two CU scenarios: (i) *low*, where the measured CU is around 15%, and (ii) *high*, where the measured CU is around 70%. We evaluate the accuracy of SATRAC

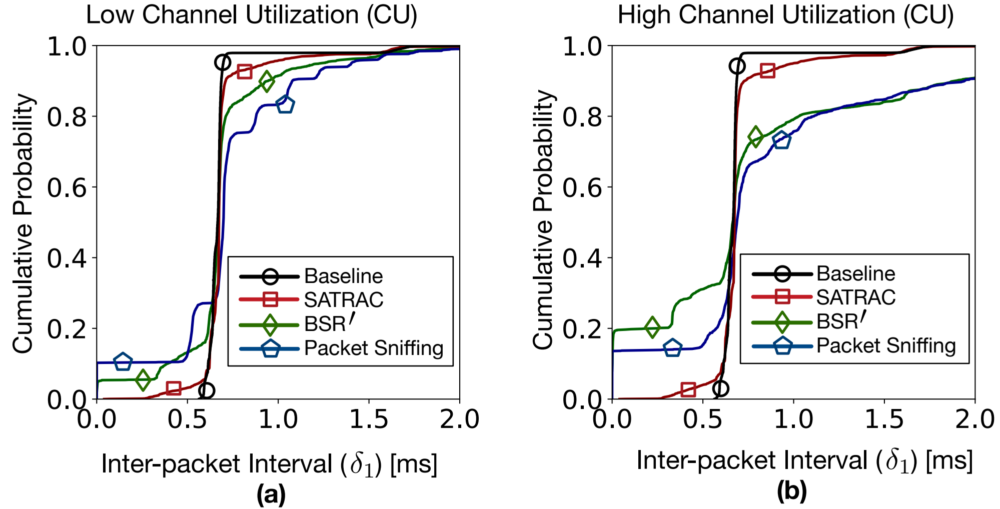

Fig. 7.8: Empirical comparison of SATRAC versus the baseline (actual data generation time instances), packet sniffing, and BSR′ in low (a) and high (b) CU scenarios. SATRAC demonstrates the highest accuracy even in the presence of high CU.

when inter-packet intervals are small. To this end, a STA runs a program that generates and sends a 1400-byte message every 500 μs. Since TCP employs a buffering delay of about 200 ms to accumulate data before transmission, we utilize the `TCP_NODELAY` flag for the TCP socket to send the generated data as soon as received from the application. Also, we are using the voice AC for the transmission of generated packets. Note that this AC does not employ packet aggregation (i.e., A-MPDU). To establish a baseline for accuracy comparison, we denote the actual data generation instances by the application as *baseline*.

Figure 7.8 presents the results collected in low and high CU scenarios. We can observe that the baseline curve is not a perfect vertical bar. The variations of the baseline are caused by multiple factors, including timer inaccuracy, context switching, the delay in copying data from the user-space to the kernel-space, and the delay in logging time stamps. The closest curve to the baseline curve is that of SATRAC, in both low and high CU scenarios, which demonstrates the high accuracy of this approach.

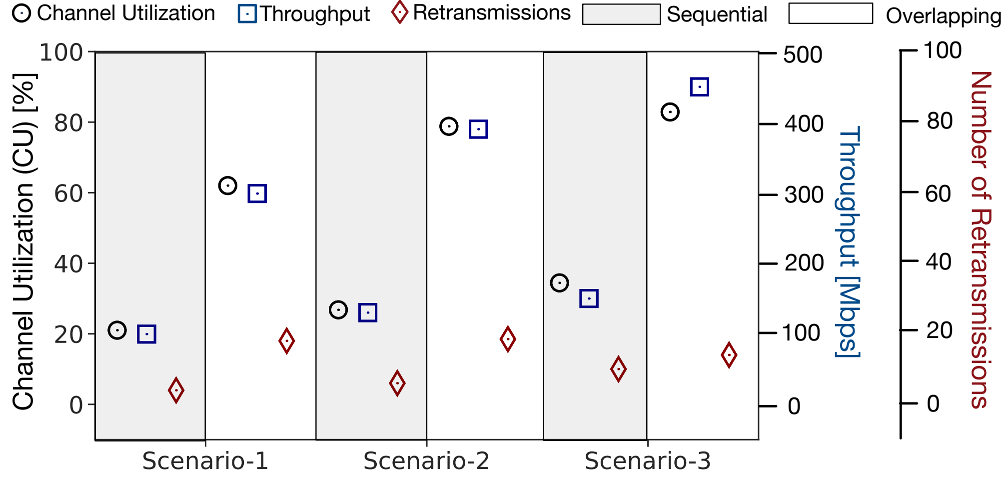

Fig. 7.9: *Sequential* and *overlapping* TWT allocations to three STAs. Throughput, CU, and number of transmissions are per second. The three scenarios refer to incrementally higher CU levels by the STAs. By leveraging SATRAC, the AP assigns a service period to multiple STAs to enhance both CU and throughput.

Note that some of the timing inaccuracies affecting the baseline also affect SATRAC; context switching delay and copying data from user-space to kernel-space are sample factors. Nevertheless, while both packet sniffing and BSR′ approaches are considerably affected in the high CU scenario, the accuracy of SATRAC remains unaffected. We also observe that although the accuracy of traffic characterization is slightly improved by using BSR′ values, the accuracy of this method is considerably affected by increasing CU.

7.4.4 Sample TWT Allocation Scenario

To show the benefits of utilizing SATRAC for TWT allocation, we use a testbed including three STAs. First, similar to the existing works [48, 37], we assign non-overlapping service periods to the STAs; we refer to this approach as 'sequential' allocation. Then, we enable the AP to characterize traffic, determine the potential for higher channel utilization, and assign overlapping service periods to the STAs; this is referred

to as ‘overlapping’ allocation. By changing the packet generation instances of each STA, we adjust inter-packet intervals and introduce various CU levels, corresponding to three scenarios. The results are presented in Figure 7.9. We observe that the overlapping allocation enhances throughput and CU, while the number of retransmissions (caused by collisions) are slightly increased. For example, in Scenario-3 where the mean per-STA CU is 36% in the sequential allocation, the overlapping allocation increases the mean to 86%. Although the overlapping allocation increases the mean number of retransmissions per second from 10 to 14, these are 0.00025% and 0.00035% of the total number of transmissions per second, respectively. Therefore, this results confirm the effectiveness of accurate traffic characterization for TWT allocation.

7.5 Summary

Allocation of TWT service periods to IoT STAs requires accurate traffic characterization to meet applications’ demands while enhancing energy efficiency and throughput. In this chapter, we empirically studied traffic burstiness and the causes of inter-packet delays in WiFi-based IoT networks. We analyzed the shortcomings of existing traffic characterization methods and introduced a novel approach based on packet modification in the source STA’s protocol stack. We showed that using eBPF to embed inter-packet generation times in TCP (or IP headers) provides an effective solution for determining per-flow traffic patterns in AP.

While in this chapter we focused on traffic characterization and TWT allocation in the time domain, the proposed method can be used to enhance the allocation of Resource Units (RUs) to STAs in 802.11ax networks. In particular, by enhancing the accuracy of conveying STAs’ demands to the AP, more efficient time and frequency (TWT and RU) allocation algorithms can be developed.

Chapter 8

Conclusion and Future Work

The works presented in this thesis addresses the challenges of (i) designing methods for accurate and high-rate data collection in WiFi networks, (ii) developing scheduling algorithms for enhancing the energy efficiency and timeliness of last-hop communication in IoT networks, and (iii) accurately determining traffic characteristics for TWT schedules in WiFi 6. This chapter presents the achievements of this research and the potential areas of future work The following summarizes the potential areas of future research:

(i) Chapter 3 proposes a framework that facilitate high-rate monitoring of the Linux networking stack. The open-sourced, publicly available MonFi tool provide two modules, i.e., the Controller, which is responsible for the receiving the statistics collection parameters from the users, whereas, the Collector is responsible of collecting the statistics across different parts of the networking stack. Furthermore MonFi tool provides three modes of statistics collection: (i) polling-based, (ii) event-based, and (iii) hybrid approach to cater to different measurement collection requirement according to applications on hand. The current implementation utilizes the Atheros ath9k driver, however, in future, this tool can easily be extended, such that it can operate with all SoftMAC-based WiFi drivers.

(ii) Chapter 4 provides a non-intrusive and highly secure framework for the monitoring of networking stack. It utilizes the eBPF technology that enables runtime patching of Linux kernel. Utilizing this framework, we extensively studied the

delays incurred by a packet across different components of the networking stack at the AP. Furthermore, we also proposed and evaluated a method of estimating the energy consumption of the WiFi subsystem of connected STAs on via the AP. We intend to extend utilize FLIP for monitoring and examining other aspects of WiFi communication, such as the causes of packet retransmissions in future.

(iii) Chapter 5 proposed a scheduling algorithm for prioritizing packets based on the remaining tail-time of the stations' PSM tail-time. The performance and applicability of the proposed approach can be enhanced by integrating context awareness. For example, detecting application layer protocol eliminates the burden of manually identifying IoT devices. Additionally, to extend the proposed mechanism to scenarios with multiple APs and mobile stations, a software-defined networking architecture can be employed to collect the required data (such as packet laxities) from multiple APs and run the proposed algorithms centrally. Another area of future work is to evaluate the reliability of communication considering different factors, such as multipath fading, MIMO, and transmission power. These dynamic factors are challenging to be described or calculated by mathematical models. Investigating energy harvesting techniques in addition to the 802.11 PSM techniques to extend the system's lifetime is left as a future work too.

(iv) Chapter 6 proposed a design for estimating the RTT of a request-response, that can be utilized by the STA's NIC to switch to low-power sleep state until the response (DL packet) is ready for delivery. Thus, conserving energy, while maintaining the minimum delay requirements. Specifically, the performance of several machine learning algorithms were evaluated with respect to the prediction accuracy of wireless transit delay considering features such as the channel utilization, MAC layer retransmission, queue statistics, etc. Extending the implementation of EAPS for next-generation WiFi 6 standard is left as future work. Additionally, the

accuracy of the delay prediction can be further enhanced by taking into consideration other variables that affect the wireless conditions in WiFi communication.

(v) Chapter 7 focuses on the improving the observability of traffic characteristics in order to accurately allocate TWT sessions in next-generation WiFi 6 standard. In particular, we evaluated common methods used for traffic characterization and discuss the shortcomings of each. We proposed a method based on the eBPF technology that can be utilized by the STA to facilitate the AP with information about inter-packet generation interval. Hence, the non-trivial task of traffic characterization can be offloaded to AP. As a future work, we intend to utilize this platform for developing novel TWT schedule allocation algorithms. In this work, we characterize traffic in three common applications in IoT domain, such a sensor that collects and sends the data to a central controller, a security camera, and a video streaming device. In future, the performance of SATRAC can also be evaluated for other IoT centric applications.

References

[1] Cisco Systems. (2020) Cisco annual internet report (2018–2023) white paper. 1

[2] A. Nikoukar, S. Raza, A. Poole, M. Güneş, and B. Dezfouli, "Low-power wireless for the internet of things: Standards and applications," *IEEE Access*, vol. 6, pp. 67 893–67 926, 2018. 1

[3] A. Bartoli, M. Dohler, J. Hernández-Serrano, A. Kountouris, and D. Barthel, "Low-power low-rate goes long-range: The case for secure and cooperative machine-to-machine communications," in *International Conference on Research in Networking*. Springer, 2011, pp. 219–230. 1

[4] 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, 2012. 1, 21

[5] D. Thomas, R. McPherson, G. Paul, and J. Irvine, "Optimizing power consumption of Wi-Fi inbuild IoT device: an MSP430 processor and an ESP-03 chip provide a power-efficient solution," *IEEE Consumer Electronics Magazine*, vol. 5, no. 4, pp. 92–100, 2016. 1

[6] 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 1, 4

[7] 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 1, 14, 60, 83, 113

[8] Silicon Labs. Zentri AMW036/AMW136 Data Sheet. [Online]. Available: https://www.silabs.com/documents/login/data-sheets/ADS-MWx36-ZentriOS-101R.pdf 1

[9] Texas Instruments Incorporated. CCC3200MOD Data Sheet. [Online]. Available: http://www.ti.com/lit/ds/symlink/cc3200mod.pdf 1

[10] F. Tramarin, S. Vitturi, M. Luvisotto, and A. Zanella, "On the use of ieee 802.11n for industrial communications," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 5, pp. 1877–1886, 2016. 1

[11] B. Dezfouli, V. Esmaeelzadeh, J. Sheth, and M. Radi, "A Review of Software-Defined WLANs: architectures and central control mechanisms," *IEEE Communications Surveys & Tutorials (COMST)*, 2018. 1, 2

[12] T. Høiland-Jørgensen, M. Kazior, D. Täht, P. Hurtig, and A. Brunstrom, "Ending the anomaly: Achieving low latency and airtime fairness in wifi," in *2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association, Santa Clara, CA*, 2017, pp. 139–151. 2, 22, 35, 50, 53, 97

[13] G. Nan, X. Qiao, J. Wang, Z. Li, J. Bu, C. Pei, M. Zhou, and D. Pei, "The Frame Latency of Personalized Livestreaming Can Be Significantly Slowed Down by WiFi," in *IPCCC*. IEEE, 2018, pp. 1–8. 2, 3, 22, 45

[14] 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. 2, 3, 21, 22, 45, 50

[15] S. Liu, V. Ramanna, and B. Dezfouli, "Empirical Study and Enhancement of Association and Long Sleep in 802.11 IoT Systems," in *Global Communications Conference (GLOBECOM)*, 2020. 2, 22

[16] 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 *Communications (ICC), 2016 IEEE International Conference on*. IEEE, 2016, pp. 1–6. 2, 14, 20, 22

[17] J. Schulz-Zander, C. Mayer, B. Ciobotaru, S. Schmid, and A. Feldmann, "OpenSDWN: Programmatic control over home and enterprise WiFi," in *ACM SOSR*, 2015, pp. 1–12. 2

[18] 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. 2, 16, 21

[19] A. J. Pinheiro, J. d. M. Bezerra, C. A. Burgardt, and D. R. Campelo, "Identifying IoT devices and events based on packet length from encrypted traffic," *Computer Communications*, vol. 144, pp. 8–17, 2019. 2, 3, 22, 45

[20] A. Sivanathan, H. H. Gharakheili, F. Loi, A. Radford, C. Wijenayake, A. Vishwanath, and V. Sivaraman, "Classifying IoT devices in smart environments using network traffic characteristics," *IEEE Transactions on Mobile Computing*, vol. 18, no. 8, pp. 1745–1759, 2018. 2, 3, 22, 45

[21] E. Anthi, L. Williams, M. Słowińska, G. Theodorakopoulos, and P. Burnap, "A supervised intrusion detection system for smart home IoT devices," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 9042–9053, 2019. 2, 3, 22, 45

[22] 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. 2, 22

[23] J. Chen, B. Liu, H. Zhou, Q. Yu, L. Gui, and X. S. Shen, "QoS-driven efficient client association in high-density software-defined WLAN," *IEEE Transactions on Vehicular Technology*, vol. 66, no. 8, pp. 7372–7383, 2017. 2

[24] H. Moura, G. V. Bessa, M. A. Vieira, and D. F. Macedo, "Ethanol: Software defined networking for 802.11 wireless networks," in *IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2015, pp. 388–396. 2

[25] 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, 2014. 2

[26] J. Sheth, C. Miremadi, A. Dezfouli, and B. Dezfouli, "EAPS: Edge-Assisted Predictive Sleep Scheduling for 802.11 IoT Stations," *arXiv preprint arXiv:2006.15514*, 2020. 2, 46

[27] F. Wilhelmi, S. Barrachina-Muñoz, 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. 2, 93

[28] A. J. Pyles, X. Qi, G. Zhou, M. Keally, and X. Liu, "SAPSM: Smart adaptive 802.11 PSM for smartphones," in *Proceedings of the 2012 ACM conference on ubiquitous computing*, 2012, pp. 11–20. 2, 5, 18

[29] MediaTek Inc. MT7687: a low power 1T1R 802.11n single-band Wi-Fi subsystem and a power management unit (PMU). [Online]. Available: https://labs.mediatek.com/en/chipset/MT7687 4

[30] Qualcomm Technologies Inc. MT7687: Low-Energy Wi-Fi Single-Band

802.11a/b/g/n SoC. [Online]. Available: https://www.qualcomm.com/products/qca4002 4

[31] E. Rozner, V. Navda, R. Ramjee, and S. Rayanchu, "Napman: network-assisted power management for wifi devices," in *Proceedings of the 8th international conference on Mobile systems, applications, and services (MobiSys).* ACM, 2010, pp. 91–106. 5, 18, 21

[32] 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. 5

[33] M. Nurchis and B. Bellalta, "Target wake time: Scheduled access in IEEE 802.11 ax WLANs," *IEEE Wireless Communications*, vol. 26, no. 2, pp. 142–150, 2019. 6, 16, 121

[34] Q. Chen and Y.-H. Zhu, "Scheduling channel access based on target wake time mechanism in 802.11 ax wlans," *IEEE Transactions on Wireless Communications*, vol. 20, no. 3, pp. 1529–1543, 2020. 6, 16, 121

[35] Q. Chen, G. Liang, and Z. Weng, "A target wake time based power conservation scheme for maximizing throughput in ieee 802.11 ax wlans," in *IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS).* IEEE, 2019, pp. 217–224. 6, 121

[36] A. Bhartia, B. Chen, D. Pallas, and W. Stone, "Clientmarshal: Regaining control from wireless clients for better experience," in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019, pp. 1–16. 6, 126

[37] C. Yang, J. Lee, and S. Bahk, "Target wake time scheduling strategies for uplink

transmission in ieee 802.11 ax networks," in *IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2021, pp. 1–6. 6, 16, 135

[38] 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. 9, 31, 48

[39] Amazon Inc. (2020) FreeRTOS: Real-time operating system for microcontrollers. [Online]. Available: https://www.freertos.org 10

[40] Amazon Inc. (2020) FreeRTOS AWS Reference Integrations. [Online]. Available: https://github.com/aws/amazon-freertos 10

[41] Microsoft Inc. (2020) Azure RTOS ThreadX documentation. [Online]. Available: https://docs.microsoft.com/en-us/azure/rtos/threadx/ 10

[42] Microsoft Inc. (2020) Azure RTOS ThreadX. [Online]. Available: https://github.com/azure-rtos/threadx 10

[43] Y. He, R. Yuan, X. Ma, and J. Li, "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. 13

[44] J. Manweiler and R. Roy Choudhury, "Avoiding the rush hours: WiFi energy management via traffic isolation," in *MobiSys*, 2011, pp. 253–266. 13

[45] 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. 14

[46] 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. 14, 118

[47] 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. 14, 118

[48] W. Qiu, G. Chen, K. N. Nguyen, A. Sehgal, P. Nayak, and J. Choi, "Category-based 802.11 ax target wake time solution," *IEEE Access*, vol. 9, pp. 100 154–100 172, 2021. 16, 135

[49] L. Liu, X. Cao, Y. Cheng, and Z. Niu, "Energy-efficient sleep scheduling for delay-constrained applications over WLANs." *IEEE Trans. Vehicular Technology*, vol. 63, no. 5, pp. 2048–2058, 2014. 16, 21

[50] 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. 16, 123

[51] B. Vijay and B. Malarkodi, "Improved QoS in WLAN using IEEE 802.11 e," *Procedia Computer Science*, vol. 89, pp. 17–26, 2016. 17, 67

[52] M. Frikha, T. Najet, and F. Tabbana, "Mapping DiffServ to MAC differentiation for IEEE 802.11e," in *Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services (AICT-ICIW)*. IEEE, 2006, pp. 79–79. 17, 67

[53] H.-P. Lin, S.-C. Huang, and R.-H. Jan, "A power-saving scheduling for infrastructure-mode 802.11 wireless LANs," *Computer Communications*, vol. 29, no. 17, pp. 3483–3492, 2006. 17

[54] Y. Li, X. Zhang, and K. L. Yeung, "A novel delayed wakeup scheme for efficient power management in infrastructure-based IEEE 802.11 WLANs," in *Wireless*

Communications and Networking Conference (WCNC). IEEE, 2015, pp. 1338–1343. 17

[55] M. Maity, B. Raman, and M. Vutukuru, "TCP download performance in dense wifi scenarios: analysis and solution," *IEEE Transactions on Mobile Computing*, vol. 16, no. 1, pp. 213–227, 2017. 17

[56] Z. Abichar and J. M. Chang, "Group-based medium access control for ieee 802.11n wireless LANs," *IEEE Transactions on Mobile Computing*, vol. 12, no. 2, pp. 304–317, 2013. 17

[57] Y. Yuan, W. A. Arbaugh, and S. Lu, "Towards scalable MAC design for high-speed wireless LANs," *EURASIP Journal on Wireless Communications and Networking*, vol. 2007, no. 1, p. 012597, 2007. 17

[58] S.-W. Kwon and D.-H. Cho, " Efficient power management scheme considering inter-user QoS in wireless LAN," in *Vehicular Technology Conference*. IEEE, 2006, pp. 1–5. 17

[59] Z. Zeng, Y. Gao, and P. Kumar, "SOFA: A sleep-optimal fair-attention scheduler for the power-saving mode of WLANs," in *31st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2011, pp. 87–98. 18

[60] F. Wamser, L. Iffländer, T. Zinner, and P. Tran-Gia, "Implementing application-aware resource allocation on a home gateway for the example of YouTube," in *International Conference on Mobile Networks and Management*. Springer, 2014, pp. 301–312. 18

[61] P. O Flaithearta, H. Melvin, and M. Schukat, "A QoS enabled multimedia WiFi access point," *International Journal of Network Management*, vol. 25, no. 4, pp. 205–222, 2015, Wiley Online Library. 18

[62] G107. The E-model: a computational model for use in transmission planning,. [Online]. Available: https://www.itu.int/rec/T-REC-G.107 18

[63] Z. A. Qazi, J. Lee, T. Jin, G. Bellala, M. Arndt, and G. Noubir, "Application-awareness in SDN," in *ACM SIGCOMM Computer Communication Review*, vol. 43, 2013, pp. 487–488. 18

[64] I. N. Bozkurt, Y. Zhou, T. Benson, B. Anwer, D. Levin, N. Feamster, A. Akella, B. Chandrasekaran, C. Huang, B. Maggs *et al.*, "Dynamic prioritization of traffic in home networks," in *CoNEXT Student Workshop, Heidelberg, Germany*, 2015. 19

[65] S. Shenker, "Fundamental design issues for the future Internet," *IEEE Journal on selected areas in communications*, vol. 13, no. 7, pp. 1176–1188, 1995. 19

[66] M. Jarschel, F. Wamser, T. Hohn, T. Zinner, and P. Tran-Gia, "SDN-based application-aware networking on the example of youtube video streaming," in *2013 Second European Workshop on Software Defined Networks (EWSDN).* IEEE, 2013, pp. 87–92. 19

[67] B. Afzal, S. A. Alvi, G. A. Shah, and W. Mahmood, "Energy efficient context aware traffic scheduling for IoT applications," *Ad Hoc Networks*, vol. 62, pp. 101–115, 2017. 19

[68] 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, 20

[69] 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

[70] K.-Y. Jang, S. Hao, A. Sheth, and R. Govindan, "Snooze: energy management in 802.11n WLANs," in *CoNEXT*, 2011, pp. 1–12. 20

[71] G. Judd and P. Steenkiste, "Fixing 802.11 access point selection," *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 3, pp. 31–31, 2002. 22

[72] J. Zhang, G. Zhang, Q. Wu, L. Song, and G. Xie, "Lazyas: Client-transparent access selection in dual-band wifi," in *26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2017, pp. 1–9. 22

[73] S. Li, M. Hedley, K. Bengston, D. Humphrey, M. Johnson, and W. Ni, "Passive localization of standard wifi devices," *IEEE Systems Journal*, vol. 13, no. 4, pp. 3929–3932, 2019. 22

[74] L. Molina, T. Kerdoncuff, D. Shehadeh, N. Montavont, and A. Blanc, "Wmsp: Bringing the wisdom of the crowd to wifi networks," *IEEE Transactions on Mobile Computing*, vol. 16, no. 12, pp. 3580–3591, 2017. 22

[75] A. Patro, S. Govindan, and S. Banerjee, "Observing home wireless experience through wifi aps," in *Proceedings of the 19th annual international conference on Mobile computing & networking*, 2013, pp. 339–350. 22

[76] Y.-C. Cheng, M. Afanasyev, P. Verkaik, P. Benkö, J. Chiang, A. C. Snoeren, S. Savage, and G. M. Voelker, "Automating cross-layer diagnosis of enterprise wireless networks," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4, pp. 25–36, 2007. 22

[77] Y.-C. Cheng, J. Bellardo, P. Benkö, A. C. Snoeren, G. M. Voelker, and S. Savage, "Jigsaw: Solving the puzzle of enterprise 802.11 analysis," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 4, pp. 39–50, 2006. 22

[78] C. Pei, Y. Zhao, G. Chen, Y. Meng, Y. Liu, Y. Su, Y. Zhang, R. Tang, and D. Pei, "How much are your neighbors interfering with your wifi delay?" in *International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2017, pp. 1–9. 22

[79] S. Biswas, J. Bicket, E. Wong, R. Musaloiu-e, A. Bhartia, and D. Aguayo, "Large-scale measurements of wireless network behavior," in *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 153–165. 22

[80] W. Li, D. Wu, R. K. Chang, and R. K. Mok, "Toward accurate network delay measurement on android phones," *IEEE Transactions on Mobile Computing*, vol. 17, no. 3, pp. 717–732, 2017. 22

[81] S. Miano, F. Risso, M. V. Bernal, M. Bertrone, and Y. Lu, "A framework for ebpf-based network functions in an era of microservices," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 133–151, 2021. 23

[82] Facebook. Facebook: Katran. [Online]. Available: https://github.com/facebookincubator/katran 23

[83] Sysdig. Sysdig falco: Behavioral activity moni-toring with container support. [Online]. Available: https://github.com/draios/oss-falco 23

[84] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th international conference on emerging networking experiments and technologies*, 2018, pp. 54–66. 23

[85] P. Neira-Ayuso, R. M. Gasca, and L. Lefevre, "Communicating between the kernel and user-space in Linux using Netlink sockets," *Software: Practice and Experience*, vol. 40, no. 9, pp. 797–810, 2010. 25, 29

[86] J. Malinen. (2020) hostapd: IEEE 802.11 AP, IEEE 802.1X/WPA/WPA2/EAP/RADIUS Authenticator. [Online]. Available: https://w1.fi/hostapd/ 31, 95

[87] S. Muramatsu, R. Kawashima, S. Saito, and H. Matsuo, "VSE: Virtual switch extension for adaptive CPU core assignment in softirq," in *IEEE 6th International Conference on Cloud Computing Technology and Science*, 2014, pp. 923–928. 36

[88] P. Mejia-Alvarez, L. E. Leyva-del Foyo, and A. Diaz-Ramirez, "Interrupt handling in classic operating systems," in *Interrupt Handling Schemes in Operating Systems*. Springer, 2018, pp. 15–25. 36

[89] C. Pei, Y. Zhao, Y. Liu, K. Tan, J. Zhang, Y. Meng, and D. Pei, "Latency-based wifi congestion control in the air for dense wifi networks," in *IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*. IEEE, 2017, pp. 1–10. 46, 50

[90] 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. 50, 52, 98

[91] J. Saldana, J. Ruiz-Mas, and J. Almodovar, "Frame aggregation in central controlled 802.11 WLANs: The latency versus throughput tradeoff," *IEEE Communications Letters*, vol. 21, no. 11, pp. 2500–2503, 2017. 50

[92] A. Showail, K. Jamshaid, and B. Shihada, "Buffer sizing in wireless networks: chal-

lenges, solutions, and opportunities," *IEEE Communications Magazine*, vol. 54, no. 4, pp. 130–137, 2016. 50, 97

[93] V. K. Ramanna, J. Sheth, S. Liu, and B. Dezfouli, "Towards understanding and enhancing association and long sleep in low-power wifi iot systems," *IEEE Transactions on Green Communications and Networking*, 2021. 57, 59

[94] M. P. Monitor, "online]: http://www. msoon. com/labequipment," *PowerMonitor/, visited Nov*, 2013. 60

[95] Compex Systems. Compex WLE900VX 3X3 MIMO wireless adapter. [Online]. Available: https://compex.com.sg/shop/wifi-module/802-11ac-wave-1/wle900vx- 61

[96] Malinen, Jouni. hostapd: IEEE 802.11ap, IEEE 802.1x. [Online]. Available: http://hostap.epitest.fi/hostapd 77

[97] R. Love, *Linux kernel development.* Pearson Education, 2010. 79

[98] R. Russell and H. Welte. Linux netfilter hacking howto. [Online]. Available: http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO 79, 96

[99] B. Hubert, G. Maxwell, R. van Mook, M. van Oosterhout, P. Schroeder, J. Spaans, and P. Larroy. Linux advanced routing & traffic control howto. [Online]. Available: https://www.tldp.org/HOWTO/Adv-Routing-HOWTO/ 79

[100] Openwrt. Network Traffic Control (QoS). [Online]. Available: https://wiki.openwrt.org/doc/howto/packet.scheduler/packet.scheduler 79

[101] R. Rosen, *Linux kernel networking: Implementation and theory.* Apress, 2014. 79

[102] M. A. Brown, "Traffic control howto," *Guide to IP Layer Network*, p. 49, 2006. 79

[103] A. Varga, "The OMNeT++ discrete event simulation system," in *Proceedings of the European Simulation Multiconference, June*, 2001, pp. 319–324. 80

[104] Cypress Semiconductor. CYW943907AEVAL1F Evaluation Kit. [Online]. Available: http://www.cypress.com/documentation/development-kitsboards/cyw943907aeval1f-evaluation-kit 83

[105] M. Version, "3.1. 1," *Edited by Andrew Banks and Rahul Gupta*, vol. 10, 2014. 84

[106] A. Bujari, G. Quadrio, C. Palazzi, D. Ronzani, D. Maggiorini, and L. Ripamonti, "Network traffic analysis of the steam game system," in *14th Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, 2017, pp. 716–719. 84

[107] C. Yu, Y. Xu, B. Liu, and Y. Liu, ""Can you SEE me now?" A measurement study of mobile video calls," in *INFOCOM Proceedings*. IEEE, 2014, pp. 1456–1464. 84

[108] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar *et al.*, "The QUIC transport protocol: Design and internet-scale deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 183–196. 85

[109] P. Kumar and B. Dezfouli, "Implementation and analysis of QUIC for MQTT," in *Computer Networks*, vol. 150. Elsevier, 2019, pp. 28–45. 85

[110] N. Soetens, J. Famaey, M. Verstappen, and S. Latre, "SDN-based management of heterogeneous home networks," in *11th International Conference on Network and Service Management (CNSM)*. IEEE, 2015, pp. 402–405. 85

[111] G. Dimopoulos, I. Leontiadis, P. Barlet-Ros, K. Papagiannaki, and P. Steenkiste, "Identifying the root cause of video streaming issues on mobile devices," in *Pro-*

ceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies, 2015, p. 24. 85

[112] 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. 85, 114

[113] S. Mangold, S. Choi, G. R. Hiertz, O. Klein, and B. Walke, "Analysis of IEEE 802.11 e for QoS support in wireless LANs," *IEEE wireless communications*, vol. 10, no. 6, pp. 40–50, 2003. 89

[114] IEEE 802.11 Working Group, "IEEE Standard for Information Technology—Telecommunications and Information Exchange Between Systems Local and Metropolitan Area Networks Specific Requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications 2007." 89

[115] S. Mangold, S. Choi, P. May, O. Klein, G. Hiertz, and L. Stibor, "IEEE 802.11 e Wireless LAN for Quality of Service," in *Proc. European Wireless*, vol. 2, 2002, pp. 32–39. 89

[116] 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 *Real-Time Systems Symposium (RTSS)*, 2013, pp. 140–149. 95

[117] 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. 96

[118] V. Jacobson, "Congestion avoidance and control," in *SIGCOMM computer communication review*, vol. 18, no. 4. ACM, 1988, pp. 314–329. 96

[119] R. Ludwig and K. Sklower, "The Eifel retransmission timer," *SIGCOMM Computer Communication Review*, vol. 30, no. 3, pp. 17–27, 2000. 96

[120] 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. 102, 103, 123

[121] 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. 103

[122] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997. 108

[123] K. M. Koudjonou and M. Rout, "A stateless deep learning framework to predict net asset value," *Neural Computing and Applications*, pp. 1–19, 2019. 108

[124] F. Chollet *et al.*, "Keras, github," *GitHub repository, https://github.com/fchollet/keras*, 2015. 108

[125] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014. 109

[126] C.-C. Li, V. K. Ramanna, D. Webber, C. Hunter, T. Hack, and B. Dezfouli, "Sensifi: A wireless sensing system for ultrahigh-rate applications," *IEEE Internet of Things Journal*, vol. 9, no. 3, pp. 2025–2043, 2022. 124

[127] L. Song, A. Striegel, and A. Mohammed, "Sniffing only control packets: A lightweight client-side wifi traffic characterization solution," *IEEE Internet of Things Journal*, vol. 8, no. 8, pp. 6536–6548, 2020. 127

List of Notations

Table 8.1: Summary of key notations and abbreviations

Notation	Meaning
Ψ	Station Awake Time
$\mathcal{E}$	Energy Consumed by Station
s_i^{iot}	An IoT station
s_i^{reg}	A regular station
$\mathbf{Q}_{net}^{iot}$	Set of IoT queues (in the qdisc layer)
$\mathbf{Q}_{net}^{reg}$	Set of regular queues (in the qdisc layer)
Q_i	An IoT queue
η	Number of IoT queues
Γ	Tail time duration
$\tilde{t}$	Last activity time of a station
$\Delta(p_i)$	Deadline of packet p_i
$\mathbf{\Delta}$	A circular queue holding $D(p_i)$ values
$\mathcal{M}(Q_i)$	Maximum tolerable deadline of queue Q_i
$\mathcal{D}(Q_i)$	Duration of transmitting packets currently in Q_i
$\mathcal{S}(Q_i)$	Packets serviced by queue Q_i during service period
$\bar{\mathcal{S}}(Q_i)$	Service size of queue Q_i
$\mathbf{Tx}$	A circular queue holding packet transmission delays
$\mu_{\mathbf{Tx}}$	Average packet transmission duration

Glossary

A-MPDU Aggregated-MAC protocol data unit

AC Access Category

ADC Analog-to-Digital Converter

AP Access Point

APSD Automatic Power Save Delivery

APSM Adaptive PSM

AU Airtime Utilization

AWS Amazon Web Services

BI Beacon Interval

BSR Buffer Status Report

CAM Continuously Active Mode

COTS Commercial Off-The-Shelf

CU Channel Utilization

DCF Distributed Coordination Function

DDoS Distributed Denial-of-Service

DIFS Distributed Inter Frame Space

DL Downlink

DMA Direct Memory Access

EAPS-E EAPS with Early wake-up

EAPS-L EAPS with Late wake-up

EAPS-M EAPS with Mid wake-up

eBPF extended Berkeley Packet Filter

ECDF Empirical Cumulative Distribution Function

EDC Event-based Data Collection

EDCA Enhanced Distributed Channel Access

EDCF Enhanced Distributed Coordination Function

EDF Earliest Deadline First

EPDC Event and Polling-based Data Collection

ETR Extra Trees Regressor

GBR Gradient Boosting Regressor

HBR Histogram-Based Gradient Boosting Regressor

HTB Hierarchical Token Bucket

IMI Inter-Measurement Interval

INT In-band Network Telemetry

IoT Internet of Things

LSTM Long Short-Term Memory

MAE Mean Absolute Error

MCS Modulation Coding Scheme

MIMO Multiple-Input Multiple-Output

MLME MAC layer management entity

MLP Multilayer perceptrons

MonFi-MNL MonFi w/ mmaped-netlink

MonFi-NL MonFi w/ netlink

MPDU MAC protocol data unit

NIC Network Interface Card

NN Neural Networks

NSM Network State Monitor

NSM-K NSM kernel-space

NSM-U NSM user-space

PDC Polling-based Data Collection

PER Packet Error Rate

PSM Power Save Mode

qdisc queuing discipline

QoS Quality of Service

QS Queue Size

QSA Queue Size All

RFR Random Forest Regressor

RNN Recurrent Neural Networks

RPi Raspberry Pi

RSSI Received Signal Strength Indicator

RTOS Real-Time Operating System

RTT Round-Trip Delay

RU Resource Unit

SATRAC Source-assisted Traffic Characterization

SMP Symmetric Multi-Processing

SP Service Period

SPI Serial Peripheral Interface

STA station

TIM Traffic Indication Message

ToS Type of Service

TWT Target Wake-up Time

TXOP Transmit Opportunity

UL Uplink

VFS Virtual File System

W-NIC Wired Network Interface Card

WL-NIC Wireless Network Interface Card

WUR Wake-up Radio

XDP eXpress Data Path