6-3-2014

Cutting Wi-Fi Scan Tax for Smart Devices

Tianxing Li
Dartmouth College

Cutting Wi-Fi Scan Tax for Smart Devices

Tianxing

Computer Science Department
Dartmouth College
Hanover, NH

Advised by Xia Zhou and Andrew Campbell

Dartmouth Computer Science Technical Report TR2014-752

June 3, 2014

Abstract

Today most popular mobile apps and location-based services require near always-on Wi-Fi connectivity (*e.g.*, Skype, Viber, Wi-Fi Finder). The Wi-Fi power drain resulting from frequent Wi-Fi active scans is undermining the battery performance of smart devices and causing users to remove apps or disable important services. We collectively call this the *scan tax problem*. The main reason for this problem is that the main processor has to be active during Wi-Fi active scans and hence consumes a significant and disproportionate amount of energy during scan periods. We propose a simple and effective architectural change, where the main processor periodically computes an SSID list and scan parameters (*i.e.* scan interval, timeout) taking into account user mobility and behavior (*e.g.* walking); allowing scan to be offloaded to the Wi-Fi radio. We design *WiScan*, a complete system to realize scan offloading, and implement our system on the Nexus 5. Both our prototype experiments and trace-driven emulations demonstrate that WiScan achieves 90%+ of the maximal connectivity (connectivity that the existing Wi-Fi scan mechanism could achieve with 5 seconds scan interval), while saving 50-62% energy for seeking connectivity (the ratio between the Wi-Fi connected duration and total time duration) compared to existing active scan implementations. We argue that our proposed shift not only significantly reduces the scan tax paid by users, but also ultimately leads to ultra-low power, always-on Wi-Fi connectivity enabling a new class of context-aware apps to emerge.

Contents

Abstract ii

Preface iii

1 Introduction **1**

2 Wi-Fi Scan Tax Problem **6**

2.1 Energy Consumption of Wi-Fi Scan 6

2.2 Impact on Wi-Fi Connectivity 9

2.3 Wi-Fi Scan Tax 11

3 WiScan: Offloading Wi-Fi Scan **14**

3.1 Concept and Design Challenges 14

3.2 Computing the Offloading SSID List 17

3.3 Configuring Scan Parameters 21

4 WiScan Prototype Evaluation **24**

4.1 WiScan Implementation 24

4.2 Energy Consumption of WiScan 26

4.3 Field Experiments 27

5 Trace-Driven Emulations **30**

5.1 Emulation Setup 30

5.2 Overall Performance 32

5.3 Efficacy of SSID Learning Schemes 34

5.4 Sensing Overhead and Performance Gain 37

6 Related Work **39**

7 Conclusion and Future Work **42**

List of Tables

2.1 Smart devices used in our measurements 7

2.2 Energy consumption and active duration of Wi-Fi radio and main processor 9

3.1 Speed range of different user motion 21

5.1 Summary of four Wi-Fi scan traces 31

List of Figures

2.1 Test bed for energy measurement 7

2.2 The overall power consumption of the phone during a single Wi-Fi scan . . 8

2.3 Energy measurements of Nexus 5 during a Wi-Fi active scan 8

2.4 Impact of the Wi-Fi scan interval on resulting Wi-Fi connectivity 10

2.5 The Wi-Fi scan tax: main processor's power draw in a Wi-Fi active scan . . 13

3.1 WiScan overview 15

3.2 Nearest-SSIDs scheme 18

3.3 Sector-based scheme 18

3.4 Non-uniform scheme 19

4.1 WiScan implementation on Nexus 5 25

4.2 Power measurement of an Nexus 5 phone during an offloaded scan in WiScan 26

4.3 Overall performance of WiScan 28

4.4 WiScan field experiments at our local city 29

5.1 Overall performance of WiScan 32

5.2 WiScan energy breakdown 33

5.3 Impact of SSID learning schemes 35

5.4 Impact of SSID list size on achieved network connectivity and energy. . . . 36
5.5 Impact of sensing on connectivity and sensing overhead. 37

Chapter 1

Introduction

Ubiquitous Wi-Fi connectivity is essential for many mobile applications running on smart devices (*e.g.* smartphones, tablets, wearable devices). It has also become increasingly important with the rise of carriers that aggressively rely on Wi-Fi to deliver cellular services, such as Republic Wireless (*1*) and FreedomPop (*2*). A key challenge to delivering ubiquitous Wi-Fi connectivity is the development of efficient Wi-Fi scanning, which allows mobile devices to connect to access points. However, the design of today's Wi-Fi scan mechanisms on smart devices results in applications draining the device's battery in unseen ways and mostly unbeknownst to smart device users.

To illustrate the impact of existing Wi-Fi scans on battery drain, consider the following simple experiments we conducted using the Nexus 4, where only Skype (Wi-Fi scans every 60s to stay connected for incoming calls), or Wi-Fi Finder app (Wi-Fi scans every 30s to seek connectivity (the ratio between connected duration and total experiment duration)) is continuously active. Running Skype or Wi-Fi Finder on the phone translates into a 43% or 60% reduction in the phone's battery life, respectively, due to the Wi-Fi scan overhead. Even when the phone has no applications running, the overhead of supporting the default

scan interval in Android framework (300*s*) on the phone is 19% of battery life. We also measured iPhone 5, and we obsevered the similar threnḍ.

The main reason for this overhead is that the main processor has to be active during the Wi-Fi scan operation and therefore consumes a significant and disproportionate amount of energy. From our measurements on a diverse range of smart devices, the main processor typically consumes 1-2 times more energy than the Wi-Fi radio during the scan operation. We refer to this energy cost of main processor as the *scan tax* paid for each Wi-Fi scan operation. There are a growing number of applications that rely on aggressive scanning for better Wi-Fi connectivity to meet the application's needs or enrich the user experience. We see this trends continuing in the future and as a result there is a critical need to find solutions that address the scan tax problem.

In this paper, we propose a simple architectural change that can significantly drive down the scan tax by offloading the Wi-Fi scan to Wi-Fi radios on smart devices. The key idea of offloading the Wi-Fi scan is as follows. First, the main processor computes a list of SSIDs and scan related parameters (*i.e.* scan interval, timeout) by taking into account Wi-Fi hotspot/AP locations and user mobility (*e.g.* walking speed) with the goal of maximizing Wi-Fi connectivity. Based on our approach, the Wi-Fi radio performs active scans independently, and only wakes up the main processor when it discovers any SSID in the SSID list. This reduces the active duration of the main processor, and improves the energy efficiency during the disconnected state. In essence, we enable low-power, aggressive hunting for Wi-Fi connectivity without involving the main processor.

To realize Wi-Fi scan offloading, however, we face several key challenges. *First*, computing the list of SSIDs to offload is tricky, because off-the-shelf Wi-Fi chipsets have very limited memory and can store only up to 10-16 SSIDs (*3, 4*), while a user may encounter

several orders of magnitude more SSIDs over a short period of time. Thus, we need intelligent algorithms to predict the right set of SSIDs that the user is likely to encounter in the near future and the device is able to connect to. The goal is that the Wi-Fi radio is able to obtain maximal Wi-Fi connectivity without waking up the main processor when there are no usable access points (*i.e.*, closed access points). *Second*, user mobility complicates the configuration of Wi-Fi scan parameters in the disconnected state, which depends on the user's current location, moving speed and direction, and the density of the surrounding public access points. *Third*, smart devices have limited computational power and battery lifetime. We need low-complexity algorithms to adapt the SSID list and scan parameters online without introducing much additional sensing overhead. In the future, the adaptation process may be fully offloaded to the Wi-Fi radio as the mobile architecture advances (*28*).

To address the above challenges, we design, implement, and evaluate *WiScan*, a complete system design that enables Wi-Fi scan offloading, and fully exploits the gain of Wi-Fi scan offloading by intelligently adapting the offloading SSID list and scan parameters based on user's mobility and behavioral patterns. To compute the offloading SSID list, WiScan uses a light-weight SSID learning algorithm that integrates historical SSIDs and user mobility prediction to estimate the SSIDs that the user will likely encounter in the near future. The key feature of our SSID learning algorithm is that it infers user mobility online without requiring continuous location sensing. WiScan represents a significant departure from the prior work on mobility prediction that typically requires frequent location sensing (*27, 31*). To configure scan parameters, WiScan integrates the output of low-power activity sensors found in smart devices to calibrate the user's velocity estimation and adapts the scan frequency and timeout based on the location of the next available hotspot.

We have built a WiScan prototype on the Nexus 5, and evaluated WiScan using both

prototype experiments and large-scale emulations driven by Wi-Fi traces collected from smartphones users across the world. Both our experiments and emulation results demonstrate WiScan's significant energy efficiency across diverse network deployment settings and user mobility patterns. It achieves at least 90% of the maximal Wi-Fi connectivity (connectivity that the existing Wi-Fi scan mechanism could achieve with 5 seconds scan interval), yet reduces the energy cost by 50-62% spent on seeking Wi-Fi connectivity.

Our key contributions are as follows:

- We identify the problem of Wi-Fi scan tax, and perform extensive measurements on popular smart devices to examine the root cause, and quantify its impact on smart device's battery life;
- We cut the Wi-Fi scan tax by offloading scans to Wi-Fi radios, and design intelligent algorithms that compute the SSID list and scan-related parameters to fully exploit the benefits of scan offloading;
- We build a WiScan prototype on the Nexus 5 smartphone by adding modules to the Android device driver, framework, and application layer, and perform real-world experiments to validate WiScan's energy saving and near-optimal connectivity;
- We evaluate WiScan using large-scale emulations driven by real traces of smartphone users in four cities across the world, and examine the impact of diverse network setups and user mobility patterns on WiScan performance.

We believe that WiScan can ultimately enable "always-on" Wi-Fi connectivity necessary for future context-aware applications, such as Glass-based augmented reality and gesture-driven HCI. All these require always-on connectivity between smart devices and the cloud. WiScan is similar in spirit to the industrial trend (*e.g.* Apple's M7, Moto X) of offloading sensing to low-power co-processors (*5, 6*), which enables always-on sensing in

support of anytime context computing. Always-on sensing with always-on Wi-Fi connectivity will allow smart devices to be more tightly connected to the cloud, resulting in new applications not possible today because of the huge drain of battery life.

Chapter 2

Wi-Fi Scan Tax Problem

Wi-Fi scan is the basic functionality to seek maximal Wi-Fi connectivity. In this section, we first examine the existing Wi-Fi scan implementation and identify the problem of Wi-Fi scan tax by measuring popular smart devices. We then analyze the impact of scan energy on maximal Wi-Fi connectivity using real-world traces from smartphone users.

2.1 Energy Consumption of Wi-Fi Scan

To examine the energy consumed by Wi-Fi scans, we test five models of Android smartphones and Google Glass (Table 2.1). We measure the device's power draw using the Monsoon power monitor (*7*), which reports instantaneous power averaged every $0.2ms$[1]. As shown in Figure 2.1, we remove the device battery, and connect battery pins to the power monitor. By powering the device using the power monitor, we are able to collect accurate power measurements. We modify the Android framework so that the radio performs scans with a specified scan interval without associating with any AP. To obtain clean

[1]We were unable to use the power monitor to measure on Google Glass. We developed an application to specify the scan interval and measured its resulting battery life.

Table 2.1: Smart devices used in our measurements. We measure the baseline power when the device is in disconnected idle state, where we turn off the phone screen, and disable all radios, apps and services.

Metric	**Smart Phone**					**Smart Glass**
	Samsung Galaxy S3	Samsung Galaxy Nexus	LG Nexus 4	LG Nexus 5	Samsung Note 3	Google Glass
OS	Android 4.1.2	Android 4.3	Android 4.4.2 (developer image)		Android 4.3	Android 4.0.4
Wi-Fi	Murata M2322007	Broadcom BCM4330	Atheros WCN3660	Broadcom BCM4339	Broadcom BCM4339	Broadcom BCM4330
Main processor	Exynos 4 quad-core 1.4GHz	TI OMAP 4460 dual-core 1.2GHz	Snapdragon S4 Pro quad-core 1.5 GHz	Snapdragon 800 quad-core 2.26GHz	Exynos 5 quad-core 1.9GHz	TI OMAP 4430 dual-core 1.2GHz
Battery	2100mAh, 3.8V	1750mAh, 3.7V	2100mAh, 3.8V	2300mAh, 3.8V	3200mAh, 3.8V	570mAh, 3.7V
Baseline power	8.87mW	18.31mW	14.04mW	12.24mW	12.70mW	23.87mW

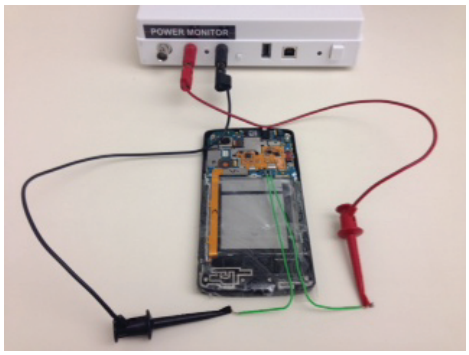


Figure 2.1: Energy measurements of Nexus 5 during a Wi-Fi active scan. We connect a Monsoon power meter to the battery pins of a smartphone. We keep only the Wi-Fi radio on, turn off the phone screen, disable all other radios and background services.

measurements, we turn off the screen, and disable all other radios and apps/services. We validate our setup by measuring each device's baseline power, where we disable all radios, apps/services with the screen off. Our baseline power measurements (Table 2.1) align with prior results (*8*).

For all measured devices, we observe that their Wi-Fi radios perform active scans when the screen is off[2]. Figure 2.2 plots the power draw of Nexus 5 during a Wi-Fi active scan.

[2]The radio conducts passive scans only when it has no SSID connection history and the screen is off. Passive scans entail long delay, where the radio stays on each channel for 400*ms*. Active scan is the dominating scan type we observed in all our measurements.

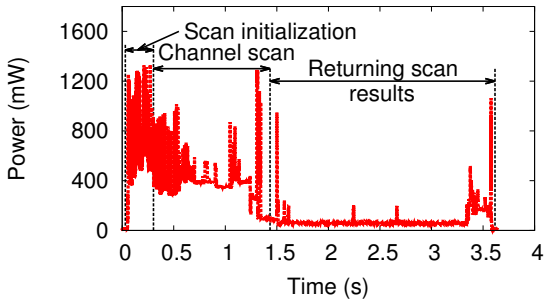


Figure 2.2: The overall power consumption of the phone during a single Wi-Fi scan.

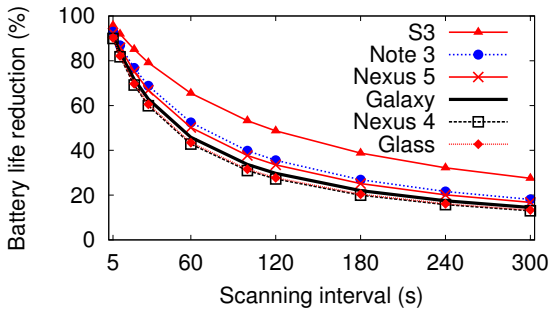


Figure 2.3: Energy measurements of Nexus 5 during a Wi-Fi active scan. It shows that frequent Wi-Fi active scans lead to significant reduction in battery life.

The scan consists of three phases: scan initialization (0 - 0.3*s*), channel scan (0.3*s* - 1.4*s*), and returning scan results to the application layer (1.4*s* - 3.6*s*). The radio first sends out a channel probe to each Wi-Fi channel to solicit replies from APs on each channel. It stays for 20*ms* on each channel, and sends the scan result to main processor. After receiving scan results, the main processor sends them to the application layer. Overall the scan lasts 3.6*s*, and consumes 0.74J energy. We observe similar patterns when measuring other devices (Table 2.2 lists detailed measurement numbers).

The energy overhead of Wi-Fi scan significantly affects the device battery life. To examine this impact, we vary Wi-Fi radio's scan interval from 5 *seconds*[3] to 300 *seconds*,

[3] 5*s* is the minimal scan interval supported by these devices because of the scan duration (3-4*s*) and channel switch delay.

Table 2.2: Energy consumption and active duration of Wi-Fi radio and main processor (MP) to finish a single Wi-Fi scan. We have subtracted the baseline power from main processor's power so that it accurately reflects the power associated with the Wi-Fi scan. Across all types of smart devices, MP consistently consumes nearly 60% of the total energy consumed by a Wi-Fi scan operation.

	Samsung S3	**Galaxy Nexus**	**Nexus 4**	**Nexus 5**	**Samsung Note 3**	**Google Glass**
Wi-Fi	0.34J (34%)	0.34J (37%)	0.26J (42%)	0.32J (44%)	0.31J (37%)	0.34J (31%)
(+bus)	0.99s	1.11s	0.85s	1.08s	1.07s	1.05s
MP	0.67J (66%)	0.59J (63%)	0.37J (58%)	0.42J (56%)	0.53J (63%)	0.76J (69%)
	2.85s	3.97s	2.16s	3.64s	3.83s	N/A

and compare the resulting battery life B^{T_s} under scan interval T_s to baseline battery life B when the device is in the disconnected idle state consuming baseline power. We derive B^{T_s} and B as follows. Assume the device has a battery with V volts and A mAh, its measured baseline power is P_b in mW, and the measured energy consumed by a Wi-Fi scan is E_s in mJ (Table 2.2, not including baseline power), then we have:

$$B = \frac{A \cdot V}{P_b}, \quad B^{T_s} = \frac{A \cdot V}{E_s/T_s + P_b}. \tag{2.1}$$

Figure 2.3 shows the battery life reduction ($1 - B/B^{T_s}$) of different smart devices under each scan interval T_s. We see that across all devices, frequent Wi-Fi scans significantly drain the batter life, causing up to 90%+ reduction in battery life. Because of the energy overhead, existing smart devices either turn off Wi-Fi radio when the screen is off or use large scan intervals (*e.g.* 300s for Android framework). Next, we will examine how different configurations of the scan interval affect the resulting Wi-Fi connectivity.

2.2 Impact on Wi-Fi Connectivity

Wi-Fi Scan Traces. We built a system service to collect Wi-Fi scan traces and user mobility. Unlike prior Wi-Fi traces (*9*) that are collected using default Wi-Fi scan intervals

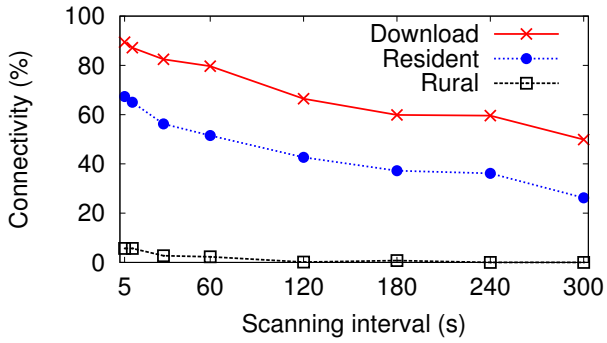


Figure 2.4: Impact of the Wi-Fi scan interval on resulting Wi-Fi connectivity, using a dataset collected in Keene, NH. Achieving maximal connectivity requires frequent scans (scan interval $<$ 60s). Default scan intervals (300s or 240s) reduces Wi-Fi connectivity by nearly half.

(180s or 300s), or using laptops, our dataset contains much finer grained Wi-Fi scan results using smartphones. This allows us to examine the impact of smaller scan interval on the Wi-Fi connectivity perceived by smart device users. Specifically, our service forces the Wi-Fi radio to scan every 5s without associating with any AP. The service collects scan results (*i.e.* SSIDs, operating channels), timestamp, GPS coordinate, and user activity inferred by Google service (*10*). We have collected Wi-Fi scan traces from four cities across the world: West Lebanon, Keene in New Hampshire, Canberra in Australia, and Beijing in China, with 4 users over 7 days. We also use the traces to drive the emulations discussed in § 5.Table 5.1 summarizes the dataset statistics.

Connectivity Results. Using these Wi-Fi scan traces as ground truth, we now examine the resulting Wi-Fi connectivity when the Wi-Fi radio scans at different frequencies. We assume the device can only connect to public SSIDs, and it connects to the SSID with the strongest received signal. The device keeps its connectivity to this SSID until it no longer sees this SSID during the scan. Then for a given scan interval and scan traces, we calculate the user's connectivity as the percentage of time that the user connects to any available public SSID.

Take the dataset at Keene as an example, which is a 5*hr* walking trace, where the user passes traffic lights, parking lots, buildings, and bridges. Figure 2.4 plots the connectivity as the scan interval varies from 5*s* to 300*s*. We identify three types of areas: downtown area with 120 APs per km^2, residential area with 40 APs per km^2, and rural area with 5 APs per km^2. We make two key observations. *First,* across all different areas, scan frequency has significant impact on the achieved Wi-Fi connectivity. A scan interval of 60*s* leads to a 20% reduction in Wi-Fi connectivity, and the default scan interval (300*s*) in today's Android framework reduces Wi-Fi connectivity by at least half. *Second*, in areas with relatively lower AP density (*i.e.* residential and rural areas), frequent scan is especially important for maximizing Wi-Fi connectivity, so that users do not miss the sparse connectivity. Overall, maximizing Wi-Fi connectivity clearly needs frequent Wi-Fi scans, which however translate into significant decrease in battery life (Figure 2.2(c)). Thus, the energy consumption of Wi-Fi scan is a big hurdle for maximizing Wi-Fi connectivity.

2.3 Wi-Fi Scan Tax

The energy consumed by the Wi-Fi scan is clearly undesirable. This begs the question: *Why does the Wi-Fi scan, a simple operation, lead to such high energy overhead?* To seek the answer to this, we examine the energy consumed by each component during an active scan. We observe that existing implementation of Wi-Fi active scan requires the main processor to be involved actively through the scan procedure. Keeping the main processor active, however, consumes considerable energy. As a result, a significant portion of scan energy is consumed by the main processor. We term the overhead of involving main processor as the *Wi-Fi scan tax*.

Quantifying Wi-Fi Scan Tax. We quantify the scan tax by measuring the main processor's power during an active scan using different phones. This is challenging because individual components (*e.g.* Wi-Fi radio, main processor) do not expose pins for us to measure their power draw. We can only measure the phone's power, which sums up the power of all components. While the Android framework provides estimated energy consumed by each component (*e.g.* display, networking), it uses the built-in battery sensor with accuracy incomparable to the power monitor.

To address this challenge, we design and implement a ghost service in the Android kernel and framework to emulate the existence of Wi-Fi scan without actually turning on Wi-Fi. The ghost service intercepts scan requests from the main processor, and sends fake scan result of each channel to the scan event handler in the framework. The timing of sending fake scan results is set based on our measurements. The main processor then processes these fake results as if the Wi-Fi radio were actually scanning, and returns the results to the application layer. We also did some energy measurement to verified that the ghost service consumes negligible energy, so the power monitor readings accurately reflect main processor's power draw.

Table 2.2 shows the energy consumption and the active duration of the Wi-Fi radio (and the bus) and main processor. We observe that across all devices, the Wi-Fi radio and the bus consume only approximately 40% of the energy for performing the scan! The majority of the energy is consumed by the main processor, which is active for 3s-4s, much longer than the radio. Clearly, existing smart devices pay a very high tax for each scan operation, and this fundamentally limits the potential of maximizing Wi-Fi connectivity (§ 2.2).

Where is the Wi-Fi Scan Tax Spent? The significant portion of Wi-Fi scan tax motivates us to analyze main processor's activities during a Wi-Fi scan. Figure 2.5 shows the

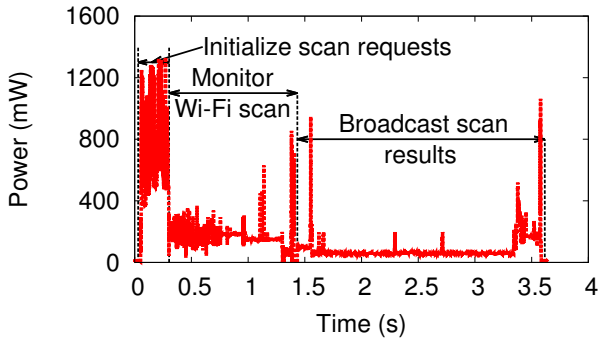


Figure 2.5: The Wi-Fi scan tax: main processor's power draw in a Wi-Fi active scan, using the Nexus 5. Similar pattern holds for other phones.

power draw of the main processor during an active scan. The main processor performs the following operations:

- *Initializing scan requests*. The main processor checks Wi-Fi radio status. If it is valid, the main processor initializes an I/O buffer space to store the package from Wi-Fi radio, and invokes Wi-Fi driver code to prepare a scan request, which includes scan type, scan interval, and scan timeout. The processor sends the scan request to Wi-Fi firmware. The peak power of these operations goes up to 1300mW.
- *Monitoring Wi-Fi scan.* The main processor monitors the scan result as the radio sends a scan probe to each channel sequentially. Whenever the radio finishes scanning a channel, the main processor collects the scan result and counts the number of APs on each channel. In the end, the main processor receives a package of scan results of all channels from the radio.
- *Broadcasting scan results.* The main processor unpacks the scan result package, which contains SSID, received signal strength, channel number, and BSSID. It then broadcasts this information to the application layer, updates UI, and waits 600+ms for an app requests. If no request comes, the processor releases the I/O buffer and sets timer for the next scan, leading to peak power up to 1000mW.

Chapter 3

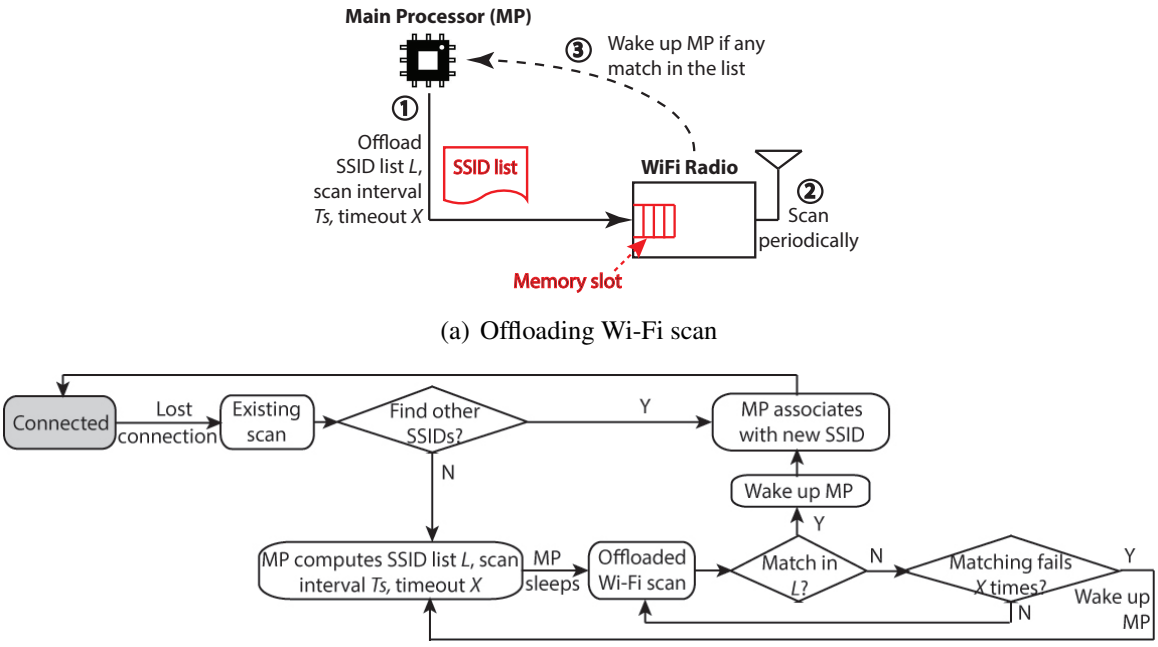
WiScan: Offloading Wi-Fi Scan

To remove the energy burden of involving main processor in the Wi-Fi scan, we propose a simple architectural change that allows the Wi-Fi radio to scan independently without requiring the main processor's participation. To fully realize the efficacy of offloading Wi-Fi scan, we design *WiScan*, a complete system that efficiently configures the SSID list and scan parameters to maximize the gain of offloading scans.

3.1 Concept and Design Challenges

WiScan centers on the concept of offloading Wi-Fi scan operation to Wi-Fi radio. This cuts the energy associated with main processor in the scan operation, and can potentially save 50-60% of scan energy, as shown in our previous measurements (Table 2.2). The device starts to offload the scan when it determines it does not currently have any available Wi-Fi connectivity and the device screen is switched off.

Figure 3.1(a) illustrates the basic concept. Before entering the sleep mode, the main processor first computes a list of SSIDs and scan-related parameters (scan interval and



(a) Offloading Wi-Fi scan

(b) Flowchart of Wi-Fi offloading

Figure 3.1: WiScan overview. (a) WiScan realizes the idea of offloading Wi-Fi scans: the main processor offloads a list of SSIDs (L) and scan-related parameters (scan interval T_s, timeout X) to the radio. The radio scans independently and wakes up main processor only when it discovers SSID in the list. (b) The system enters the mode of Wi-Fi scan offloading in the disconnected state.

timeout), and writes the SSID list into the memory slot of the Wi-Fi radio. The SSID list contains the SSIDs and their encryption information (*e.g.* encryption type, password). The Wi-Fi radio then performs active scans with a fixed scan interval and timeout. At the end of each scan operation, the radio compares its discovered SSIDs to the SSID list in its memory. If any match is found, the radio wakes up the main processor to further associate with the matched SSID. If there are multiple SSID matches, the SSID with the strongest signal will be picked. Here the SSID list is meant to filter out private APs that the device cannot connect to. This ensures that the radio wakes up the main processor only if Wi-Fi connectivity is available. The radio also wakes up the processor if it cannot find any matches after X independent scans. The timeout value allows the system to re-adjust its

SSID list and scan parameters based on current situation (Figure 3.1(b)).

Design Challenges. When realizing the concept of offloading Wi-Fi scan, we face two key design challenges: 1) determining the offloading SSID list so that the device does not miss any available Wi-Fi connectivity; and 2) determining the scan frequency and timeout to avoid unnecessary scan operations and thus save energy. Configuring these parameters brings three additional challenges. *First*, off-the-shelf Wi-Fi chipsets have very limited memory, and can only store up to 10-16 SSIDs (*3, 4*). Currently, the cost of increasing the memory store in the chipset is very costly. Since the number of usable SSIDs that a device encounters can be multiple orders of magnitude higher, it is tricky to decide the list of SSIDs to write in the memory so that the device does not miss any available Wi-Fi connectivity. *Second*, user mobility further complicates the configuration. We need to predict user location in the near future and thus the SSIDs that this user is likely to encounter. While there have been considerable amount of work on mobility prediction (*27, 31*), it typically requires training using long-term historical data, and assumes that users have regular routine. These assumptions, however, do not always hold (*e.g.* users travel to new places). We need to predict user mobility on the fly even without much historical data. *Third*, smart devices have tight energy constraints and limited computation power. So for any algorithms to be practical, they have to run in low complexity with low energy. This implies that the algorithm should be selective on the required sensor input.

Next we describe our solutions to configuring SSID list and scan parameters to address the above challenges.

3.2 Computing the Offloading SSID List

Our first key design component is to determine the SSID list to offload to the Wi-Fi radio, so that if any usable SSIDs are found, the radio will wake up main processor to further connect to the SSID. Since public hotspot locations are available in public databases such as JiWire (*11*), we assume they are known to the system as a priori. The key challenge comes from the significant discrepancy between the limited number of SSIDs the radio can store, and the large number of SSIDs the device can encounter. Let N be the maximal number of SSIDs Wi-Fi memory can store. We aim to seek schemes that generate list L of N SSIDs containing the next SSID that the device will encounter and can connect to.

Our search starts with a few straw-man solutions. The simplest approach is *popular SSIDs*, where we offline configure L to be a set of the most popular SSIDs in the user's current location. To examine how well this approach works, we crawled large-scale hotspot data[1] from Jiwire for three cities: Seattle, Chicago, and San Francisco. Our analysis shows that popular SSIDs (occurrence $> 1\%$) occupy only 10+% of the SSIDs at each city (4.8% for San Francisco). We also estimate the area covered by popular SSIDs, assuming each hotspot covers a circle area.

We tried different coverage radius from 50*m* to 200*m*. We found that popular SSIDs covers only 40%+ of all SSIDs' coverage. Our tests further show that solely relying on a few popular SSIDs leads to significant loss (>50%) of Wi-Fi connectivity.

The above observations turn our attentions to approaches of computing L online. An intuitive approach is to select the nearest N SSIDs based on user's current location, referred to as the *nearest-SSIDs* scheme. We assume that we in this project, we can get the

[1]Our dataset contains SSIDs and GPS coordinates for 1307, 1005, and 972 hotspots at city and suburban areas of Seattle, Chicago, and San Francisco, respectively. We use Google Geocoding API (*12*) to obtain hotspot GPS coordinates based on their addresses on Jiwire.

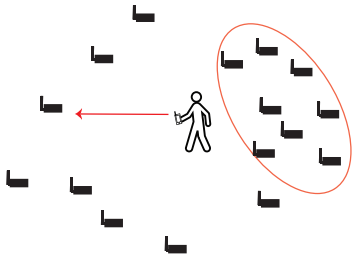


Figure 3.2: Nearest-SSIDs scheme: Schemes of deciding SSID list assuming $N = 8$, with selected SSIDs marked by red circles. Nearest-SSIDs scheme is highly affected by AP distribution. A user can be stuck with SSIDs that he/she is moving away from and unable to connect to

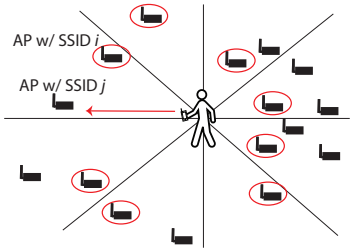


Figure 3.3: Sector-based scheme: it selects SSID in all directions. But an SSID per sector still can lead to missed connectivity. Here the user will encounter SSID_j, rather than the selected SSID_i.

location of all public SSID list from third-party sources like JiWire. The assumption is that the distance to an SSID dictates when the next Wi-Fi connectivity will be available. While simple, this approach ignores the user's moving direction and can result in incorrect prediction of the next available SSID. Figure 3.2 illustrates a simple example, where the user's nearest N SSIDs are clustered. The user is walking aways from the coverage and can no longer connect to any of them of the SSIDs. In this case, the nearest SSIDs scheme still predicts these SSIDs, which clearly are no longer relevant to the user.

Sector-based Scheme. To take into account user's moving direction, a smarter alternative is to divide all nearby SSIDs into N equal sectors and pick the nearest SSID in each sector, referred to as the *sector-based* scheme (Figure 3.3). The rationale is that since the N selected SSIDs are spread in all directions, at least one of them will be encountered regardless of the user's moving direction. This scheme is simple, and does not require mobility

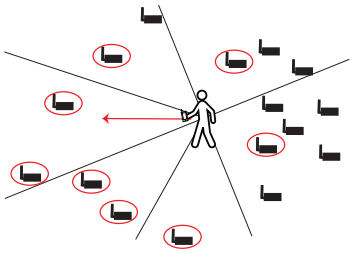


Figure 3.4: Non-uniform scheme picks more SSIDs in the estimated user's moving direction, and thus better predicts the SSID the user will encounter.

estimation. But it is limited by the small number of sectors (*i.e.* the number of SSIDs N), each with a wide angular range. As a result, a single SSID cannot cover all possible traces taken by the user in this sector, leading to connectivity losses (Figure 3.3).

Non-uniform Scheme. An optimized scheme is a variant of the sector-based scheme, where we perform non-uniform partition of the sectors guided by the estimated user's moving direction, and select more SSIDs located in sectors aligning with user's moving direction (Figure 3.4). Furthermore, we also consider previously connected SSIDs, which serve as a valuable indication of future possible Wi-Fi connectivity, especially for users with regular mobility pattern. Our approach takes the following input: 1) the user's current location, and 2) map information (*e.g.* street). Obtaining 1) need online location sensing using GPS, and we will examine the sensing overhead in § 4.3 and § 5.4. We will also examine the contribution of including historical SSIDs and map information in Figure 5.3. Specifically, our approach has the following three steps.

Step #1: Predicting Direction. We predict user's moving direction if the activity inference is non-stationary. Assume p_{t_i} is user's current location at time t_i, we estimate user's moving direction by the vector $\overrightarrow{p_{t_{i-1}} p_{t_i}}$ from the previous saved location point $p_{t_{i-1}}$ to the current location p_{t_i}. Here $p_{t_{i-1}}$ is the latest location point that is at least $10m$ away from p_{t_i} and $t_i - t_{i-1} > 60s$. This is to reduce the impact of potential location sensing error (*e.g.*

GPS or Wi-Fi localization error can go up to 10*m*+). We further refine the direction estimation by using street/road information to eliminate directions where feasible paths do not exist. A key benefit of our prediction method is that it does not require intensive location sensing, and works upon sparse historical location data saved when the device has Wi-Fi connectivity.

Step #2: Partitioning Sectors. Based on the estimated moving direction $\overrightarrow{p_{t_{i-1}}p_{t_i}}$, we first divide the space around the user into N uniform sectors, such that $\overrightarrow{p_{t_{i-1}}p_{t_i}}$ is the angular bisector of a sector. We classify the N sectors into two groups: forward sectors whose angular bisectors have acute angles to $\overrightarrow{p_{t_{i-1}}p_{t_i}}$, and backward sectors whose angular bisectors have obtuse angels to $\overrightarrow{p_{t_{i-1}}p_{t_i}}$. Then we adjust the sector partition by merging every x adjacent backward sectors, resulting in $\lfloor N/(2x) \rfloor$ backward sectors. We exhaustively tested different x values offline, and $x = 2$ performs the best in all experiments. Since we assign more sectors in forward direction, we enhance the likelihood of picking SSIDs the user will encounter. We still consider backward sectors due to the observed non-uniformity of Wi-Fi signal propagation: The device cannot connect to any AP in backward sectors now, yet it can as it moves to a direction with better received signal. In the end, we have $M = \lceil N/2 \rceil + \lfloor N/4 \rfloor$ sectors. In Figure 3.4, it shows a simple example, where 4 backward sectors are merged into 2 sectors, resulting to 6 sectors in total.

Step #3: Selecting SSIDs. The last missing piece is to select N SSIDs from these M sectors. A straightforward method is to use the distance as the only metric, which, however, can fail to provide accurate estimation because of the complex wireless propagation. Instead, we integrate it with the information of previously connected SSIDs. Our selection works as follows. We first pick $2N$ nearest SSIDs as the pool of candidate SSIDs. This is similar in spirit to the ghost list design in adaptive caching algorithm (ARC) (*25*). Let

Table 3.1: Speed range of different user motion

	User Speed (m/s)			
	Still/Tilting	Walking	Biking	Driving
v^{min}	0	0.5	2.8	5.6
v^{max}	0	1.5	13.9	36.1

$S^{history}$ denote the subset of SSIDs connected before, which are sorted by the number of times they have been connected, and S^{new} denote the subset of new SSIDs, sorted by their distances to p_{t_i}. We then project the SSIDs in $S^{history}$ into the M sectors, and select the top-ranked history SSID from each sector. For sectors without any history SSIDs, we pick the nearest SSID in each of these sectors. Finally, we pick the remaining closest $(N - M)$ SSIDs from S^{new} located in forward sectors, and this completes the SSID selection.

3.3 Configuring Scan Parameters

The second key design component is to determine the frequency and timeout of offloaded Wi-Fi scan. The challenge is to balance the tradeoff between Wi-Fi connectivity and energy consumption. Maximizing Wi-Fi connectivity needs frequent scans (Figure 2.4), which however can lead to energy waste when no usable SSIDs are discovered. We address this challenge by adapting the scan interval and timeout based on nearby hotspot distribution and user's mobility pattern. The goal is to estimate when the next available Wi-Fi connectivity will occur, and adjust the scan interval T_s and timeout X accordingly based on the estimation of user's mobility.

Adapting Scan Interval. Our method uses the following input: 1) user current location, and 2) activity inference from Google Play Service using low-power sensors (*e.g.* gyro sensor and accelerometer) (*10*). It works as follows. *First*, we obtain user's current motion status from activity inference, which classifies user's activity into five types: still, tilting,

walking, biking, and driving. If the user is static and cannot connect to any SSID, Wi-Fi radio scans with the maximal scan interval $1000s$ that the hardware can support and we continue to monitor the activity to detect user's activity changes. If the user is in one of the latter three non-stationary statuses, we estimate the moving velocity $\vec{v}$ using the current location p_{t_i} and the previous location record $p_{t_{i-1}}$, where $\vec{v} = |p_{t_i} - p_{t_{i-1}}|/(t_i - t_{i-1})$. To reduce the sensitivity to location estimation errors and the sparsity of historical location data, we calibrate $\vec{v}$ using the speed range inferred by the user's current activity status, as shown in Table 3.1 (*14, 23*). If $\vec{v} < v^{min}$, or $\vec{v} > v^{max}$, we calibrate $\vec{v}$ to v^{min} or v^{max} respectively. Otherwise we do not adjust its value. *Second,* we use street information to infer the feasible path from the current location to the nearest SSID in L, and estimate $\widetilde{T}_s$, the time for the next scan, by dividing the inferred path length over the estimated velocity $\vec{v}$. Overestimating the scan interval will make the system miss potential connectivity until the next scan. Thus, we configure scan interval T_s conservatively. We map $\widetilde{T}_s$ into pre-defined time windows: $0s - 10s$, $10s - 40s$, $40s - 70s$. We set the final scan interval T_s as the minimal value of the time window $\widetilde{T}_s$ resides.

Configuring Timeout Value. The timeout value X is set to prevent the system from sticking with an outdated SSID list and missing new connectivity. Thus, if Wi-Fi radio fails to find any SSID in the list after X offloaded scans, the radio wakes up the main processor to re-compute an SSID list. Because of the energy cost of computing an SSID list, we need to reduce the number of SSID list updates while ensuring that the list is still relevant to the current network environment. In WiScan, we calculate X by estimating the duration for the user to step out of the coverage of all SSIDs in the list. Specifically, we leverage the estimated moving direction in Non-uniform, and derive the probability p_i of the user moving towards sector i assuming p_i follows a standard normal distribution, where

the sector in the estimated direction has the highest probability. Then the expected distance $\overline{d}$ to the furthest SSID is calculated as $\overline{d} = \sum d_i * p_i, 1 \leq i \leq N$, where d_i is the distance to the furthest SSID in sector i. The timeout value X is then set as $\overline{d}/d_{min}$, where d_{min} is the distance to the nearest SSID in the list. This reflects the rough number of scans before the user steps out of the coverage of the current SSID list.

Chapter 4

WiScan Prototype Evaluation

We build a proof-of-concept prototype of WiScan and examine its performance using real-world experiments. We describe the implementation details of WiScan, and summarize our experiment findings on WiScan's energy consumption and achieved Wi-Fi connectivity.

4.1 WiScan Implementation

We implement WiScan on the Nexus 5 Android phone (4.4.2 OS developer image) that provides necessary hardware-level support for offloading Wi-Fi scan. The phone uses the Broadcom BCM4339 Wi-Fi chipset, where the Wi-Fi scan operation is built in the chipset's firmware, and the chipset has a small memory slot that can store up to 16 SSIDs. This Wi-Fi chipset supports a mode called "scheduled scan". In this mode, when Wi-Fi radio is first switched on, the framework reads an SSID list from the `wpa_supplicant.conf` file under `/data/misc/wifi/`, and loads these SSIDs to Wi-Fi radio's memory. The radio performs active scans with a fixed scan interval ($15s$)[1] independently, and wakes up the

[1] In Android 4.4.2 factory image, the scan interval starts from $15s$, and then gradually doubles after 4 failed tries for each interval capped by $240s$.

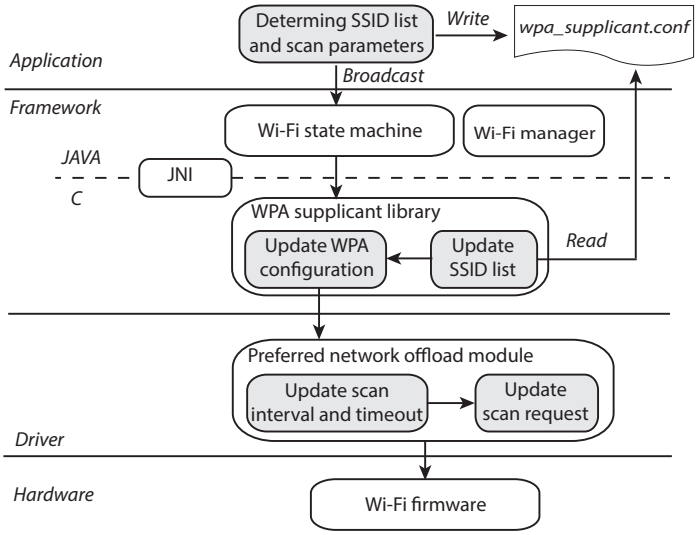


Figure 4.1: WiScan implementation on Nexus 5 (Android 4.4.2 developer image). Shaded blocks are modified/added modules. SSID list configuration is implemented above the driver level, and scan parameters (scan interval, timeout) are adapted in the driver.

main processor only when it discovers any of these 16 SSIDs. The SSID list is configured as the 16 most recently connected SSIDs. Both the SSID list and scan interval are static and non-configurable. This scheduled scan mode, however, is disabled in the developer image.

To implement WiScan, we need to activate the scheduled scan under specified conditions, and realize real-time configuration of the SSID list and scan parameters using proposed algorithms (§ 3.2 and § 3.3). We accomplish these tasks by modifying and adding related modules at the driver, framework, and application levels. Figure 4.1 shows the overall system architecture of WiScan implementation. Specifically, we implemented SSID list configuration using modules above the driver layer. Our algorithms are implemented at the application layer as a background system service. It writes a new SSID list in `wpa_supplicant.conf` file, which will be read by our added module in the WPA supplicant library at framework. Our module uses the new SSID list to replace the previous list in WPA configuration data structure. The configuration of scan parameters is implemented

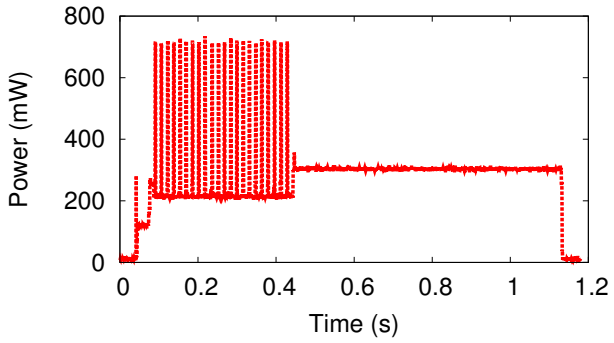


Figure 4.2: Power measurement of an Nexus 5 phone during an offloaded scan in WiScan. By offloading the scan operation to the Wi-Fi radio, WiScan removes the scan tax. A single scan consumes 0.33J energy in total (down from 0.74J of the existing scan in Table 2.2).

in the driver. We rewrote the Wi-Fi driver so that it can create a new scan request using specified SSID list and scan parameters, and pass the request to Wi-Fi firmware. To avoid the delay ($\approx 1s$) of cleaning cached scan requests, we set up a parallel thread to expedite scan offloading, which cleans cached requests while generating a new request.

4.2 Energy Consumption of WiScan

We start with examining the energy consumption of an offloaded Wi-Fi scan in WiScan. We follow the measurement set up in Figure 2.1, and plot in Figure 4.2 the instantaneous power draw of the phone during an offloaded scan. Comparing it to the existing scan (Figure 2.2), we make the following observations. *First*, offloading scan significantly shortens the scan duration by removing the main processor's operations. An offloaded Wi-Fi scan lasts $1.1s$, down from $3.6s$ of existing Wi-Fi scans. Specifically, it has two phases: 1) the radio performs active scan on 22 channels, generating 22 energy spikes up to 700mW+ each lasting 1.7-2ms (700mW+ stays for $1ms$). The radio will compare the scan result of each channel to the SSID list in the memory, and identify matched SSIDs; 2) the radio will

release the chipset buffer storing scan results, and prepare for the next scan. This results into 300mW+ power draw.

Second, the offloaded Wi-Fi scan consumes 56% less energy by removing the energy portion consumed by the main processor. Now the total energy of a scan only comes from the Wi-Fi radio (0.32J) plus the phone's baseline energy (0.01J). This matches our previous energy measurements on Nexus 5 (Table 2.2), and we expect similar energy reduction (60%+) for other platform like iOS and Windows Phone.

We note that WiScan does introduce energy overhead for calculating SSID list and scan parameters. The overhead is from: 1) the main processor running configuration algorithms (computing SSID list and scan parameters), 2) the GPS for the user's current location, and 3) the activity sensors for Google activity inference. To quantify the overhead, we measure each component's energy by instrumenting the phone to perform each operation. Our measurements show that the main processor consumes 0.1J to run our algorithms, GPS on Nexus 5 consumes 0.7J to return a location, and activity sensors cost 0.1J to infer activity. Note that WiScan does not pay all the above overhead for each offloaded scan, rather, the overhead occurs only when the system first enters the mode of scan offloading, or when offloaded scans fail to find any SSID matches after hitting the timeout. Thus the cost is shared by multiple offloaded scans. We will examine the energy consumed by the system in § 4.3 and § 5.2.

4.3 Field Experiments

Next we test WiScan using real-world experiments in a shopping area at West Lebanon, NH. The area is roughly $4km^2$, with 75 APs (44 SSIDs) set up by commercial stores, cafes, and restaurants. 5 SSIDs are encrypted, and we requested passwords from their owners. All

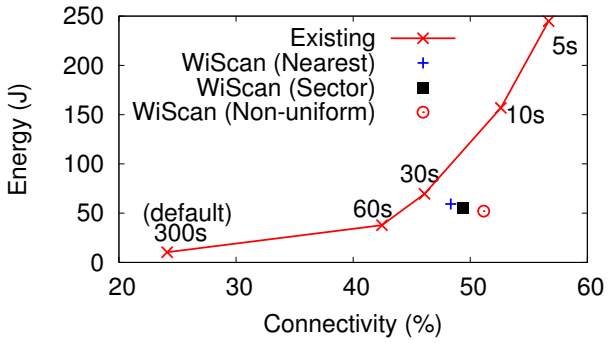


Figure 4.3: Overall performance of WiScan

SSID information is saved on the phone. In the experiments, a user walked around with occasional pauses while carrying three Nexus 5 phones. One of the phones is implemented with WiScan, and the others use existing scan with different fixed scan interval. We logged the activity and duration of GPS, activity sensors, main processor, and Wi-Fi radio. We repeated the experiment in three rounds, and tested existing scan with fixed scan interval of 5*s* (the minimal), 10*s*, 30*s*, 60*s*, and 300*s* (the default). Each round lasts 2*hr* with the same walking route.

We compare WiScan and the default existing scan for Wi-Fi connectivity (*i.e.* the percentage of connected time), and the energy consumed in the disconnected state for seeking Wi-Fi connectivity. Figure 4.3 compares existing scans under different scan intervals, to WiScan with three SSID algorithms. Our key observation is that WiScan significantly reduces the energy cost while achieving similar connectivity. Compared to existing scans with the highest frequency ($T_s = 5s$), WiScan achieves 90% of its connectivity, using only 21% of its energy. WiScan achieves similar connectivity to existing scan with $T_s = 10s$, yet reduces the energy by two thirds! To understand the source of the energy saving, we further examine the energy consumed by each component, and compare WiScan to the existing scan with similar connectivity ($T_s = 10s$). As shown in Figure 4.4, cutting the scan

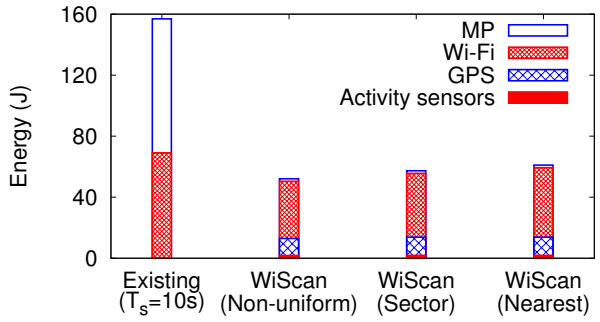


Figure 4.4: WiScan field experiments at our local city. We held multiple Nexus 5 phones, where one phone is implemented with WiScan, and the others use existing scans under different scan intervals. We observe that WiScan reduces 67% energy compared to that consumed by existing scan to achieve similar connectivity.

tax is the main source of the gain, which occupies 56% of the energy for existing scan, yet only 3% in WiScan. Another source of gain is adapting the scan interval and timeout. This avoids unnecessary scans when no hotspots are nearby, reducing the radio's energy consumption. We also notice that sensing (GPS and activity sensors) adds minor energy overhead with GPS as the dominating factor (22% of overall energy). This, however, can be driven down with recent low-power GPS design (*21*). In addition, Non-uniform moderately outperforms other SSID schemes. By leveraging the estimated user's moving direction, it predicts future SSIDs more accurately, leading to 4-6% increase in connectivity compared to other schemes. Overall, our results validate that WiScan effectively cuts scan tax with efficient configurations of SSID list and scan parameters.

Chapter 5

Trace-Driven Emulations

After examining WiScan in prototype experiments, we use large-scale, trace-driven emulations to perform more detailed evaluations on WiScan. Our emulations use mobility traces of smartphone users with different mobility patterns in different cities. This allows us to evaluate WiScan under diverse network deployments and user mobility patterns. Specifically, we seek to understand WiScan's performance in different network setups, the performance perceived by users with different mobility patterns, and the impact of design choice for each component in WiScan.

5.1 Emulation Setup

We developed an emulator using Python to examine WiScan and existing scan implementation. The emulator takes Wi-Fi scan traces described in § 2.2 as the ground truth of Wi-Fi connectivity at each location. We obtain hotspot locations using both public hotspot database JiWire, and manual labeling. Table 5.1 summarizes dataset statistics for hotspots with known locations. The datasets cover diverse hotspot density and are collected by users

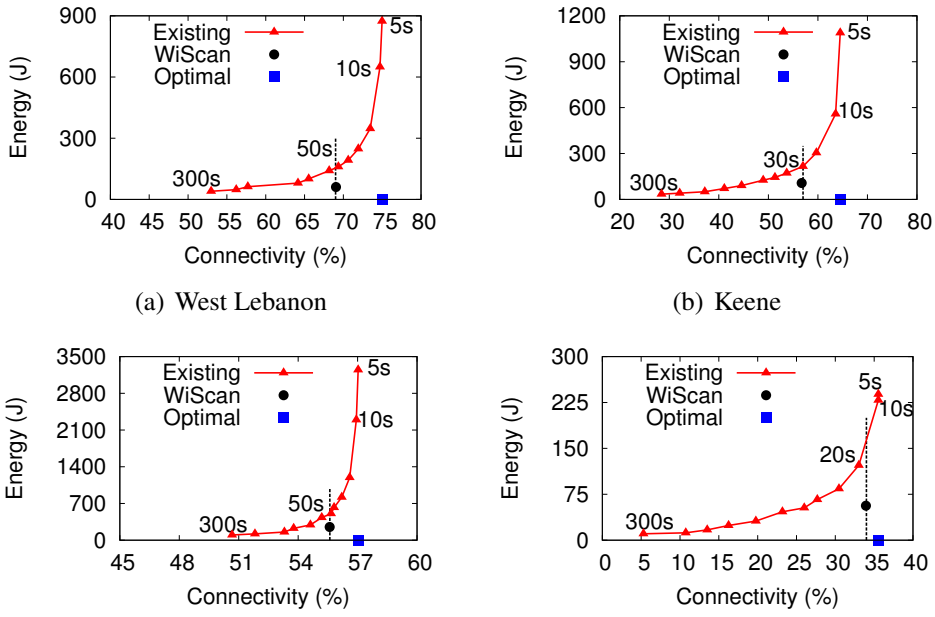
Table 5.1: Summary of four Wi-Fi scan traces, collected from smartphone users at four cities across the world.

Dataset (City)	# of public SSIDs	Area	Duration	% of mobility duration
West Lebanon	44 (75 APs)	$4km^2$	10.5*hr*	81%
Keene	148 (183 APs)	$6km^2$	5*hr*	73%
Canberra	72 (241 APs)	$9km^2$	21.8*hr*	12%
Beijing	41 (41 APs)	$2km^2$	4*hr*	37%

with different mobility regularity. Users in the West Lebanon, NH and Canberra datasets follow a regular mobility pattern: they mostly commute between home and work places with repeated routes. The mobility traces of Keene and Beijing datasets do not contain repeated routes, representing cases when users travel to new places.

Focusing on public hotspots, we assume that the device can connect to an SSID if the device can discover this SSID during its scan and its received signal strength is above about -90dBm. Based on our prototype experiments, we assume 4*s* delay for associating with an AP, and 3*s* delay for obtaining location from GPS sensor. We follow the WiScan flowchart (Figure 3.1(b)) to emulate how WiScan behaves using the Wi-Fi scan traces. We consider existing scan implementation, which entails scan tax, as the baseline referred to as *Existing*. Using Existing, Wi-Fi radio scans with a fixed scan interval in the disconnected state. We examine Existing with fixed scan interval from the minimal (5*s*) to the default (300*s*).

We evaluate WiScan and Existing in both achieved connectivity and energy cost. Similarly as § 4.3, we define connectivity as the percentage of connected time, which reflects the actual performance perceived by end users given their mobility pattern. We define energy cost as the energy consumed in the disconnected state for seeking Wi-Fi connectivity. We calculate the energy cost based on the measurement of WiScan (§ 4.2) and Existing on Nexus 5 (Table 2.2).



(a) West Lebanon (b) Keene

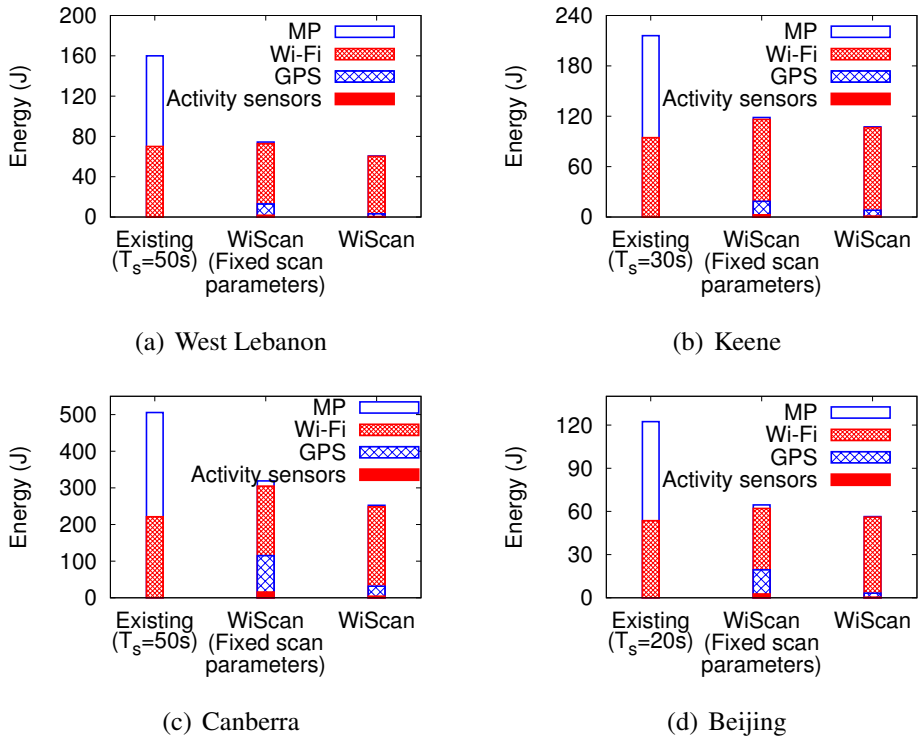
(c) Canberra (d) Beijing

Figure 5.1: Overall performance of WiScan, compared to existing scan implementation with different scan interval. WiScan achieves 90%+ of the optimal connectivity, while reducing 50%-62% of the energy cost for seeking Wi-Fi connectivity compared to existing scan with similar connectivity.

5.2 Overall Performance

We start with evaluating WiScan's overall performance compared to Existing using different dataset. We also calculate the optimal connectivity assuming perfect knowledge on future Wi-Fi connectivity. The optimal connectivity is the percentage of connected duration when the device never misses any hotspot connectivity. Figure 5.1 compares WiScan to Existing with different fixed scan rate and the optimal. Our key observations are as follows.

Observation #1: WiScan achieves at least 90% optimal connectivity. Across all four datasets, WiScan consistently achieves at least 90% of the optimal connectivity. This demonstrates that the our SSID learning scheme accurately predicts future SSIDs, hence the system does not miss available Wi-Fi connectivity while allowing the main processor to



(a) West Lebanon (b) Keene

(c) Canberra (d) Beijing

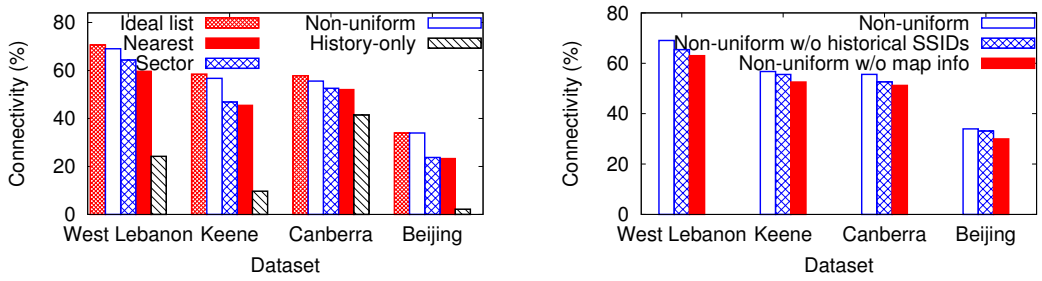
Figure 5.2: WiScan energy breakdown, compared to WiScan without adaptive scan parameters (*i.e.* scan interval, timeout) and existing scan. We configure the scan intervals of the latter two so that they achieve nearly the same connectivity as WiScan. WiScan's significant energy saving is from cutting the scan tax (energy consumed by main processor), and adapting scan parameters to reduce the number of SSID list computations.

sleep when no public hotspot is nearby. In particular, WiScan achieves 97% of the optimal connectivity for the Canberra dataset, because its trace has low mobility (Table 5.1) and follows regular pattern (work and home). Non-uniform makes more accurate prediction in this case. We notice that although the Beijing's dataset has high hotspot/AP density (20 per km^2), its absolute connectivity percentage is lower than other datasets. This is because the hotspots in this dataset are distributed unevenly, where the majority of SSIDs are clustered in a small area and their coverage areas significantly overlap. Thus the overall high hotspot density does not translate into high percentage of connected time.

Observation #2: WiScan reduces 50-62% energy cost of seeking Wi-Fi connectivity compared to existing scans. While achieving connectivity close to the optimal, WiScan significantly reduces the energy cost for seeking Wi-Fi connectivity. As shown in Figure 5.1, WiScan reduces 50%-62% of the energy spent on seeking Wi-Fi connectivity, compared to Existing with the scan interval achieving similar connectivity. In Figure 5.2, it further shows each component's energy. We include WiScan with fixed scan interval and default timeout value (4) to understand the gain of adapting scan parameters. Our observations are as follows. *First*, cutting the scan tax contributes a significant portion of energy saving. While the main processor consumes more than half of the scan energy for Existing, it consumes less than 1% of energy in WiScan. This indicates that computing SSID list and scan parameters entails negligible energy cost on the main processor. *Second*, adapting scan parameters contributes a moderate reduction (10-30%) in energy cost. The adaptation achieves higher gain in the Canberra and Beijing dataset, where available connectivity is lower than other datasets (Figure 5.1(c)(d)), and users are mostly stationary (88% and 63% of the total duration respectively as shown in Table 5.1). In this case, fixing scan parameters leads to a larger number of unnecessary scans, which can be avoided by adapting scan parameters. *Third*, the sensing overhead of WiScan is negligible, with GPS sensor as the dominating sensor occupying 87.5% of the overall sensing overhead. The energy saving of cutting Wi-Fi scan tax greatly outweighs the sensing overhead of WiScan. Overall, our results verify the efficacy of WiScan under diverse network environment and user mobility.

5.3 Efficacy of SSID Learning Schemes

After examining the overall performance and the contribution of adaptive scan parameters, next we evaluate different SSID learning schemes, design choices within the Non-uniform



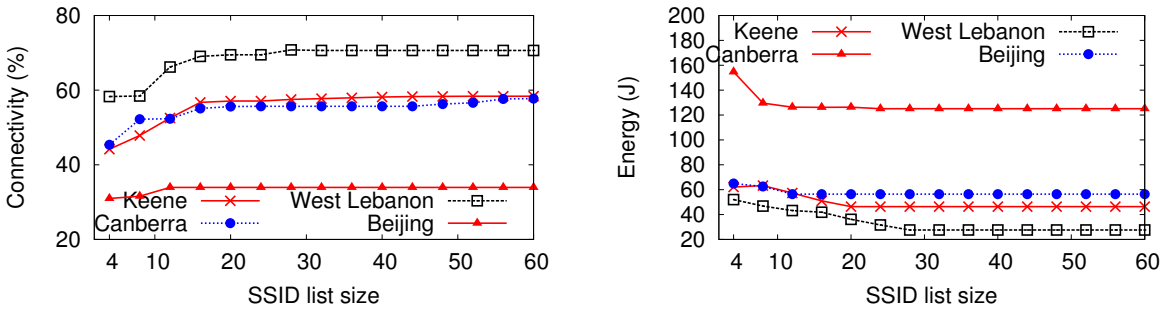
(a) SSID schemes

(b) Non-uniform

Figure 5.3: Impact of SSID learning schemes, decision choices within Non-uniform. Non-uniform consistently achieves 96%+ of the connectivity of an ideal SSID list, and outperforms other solutions under all datasets.

scheme, and the impact of SSID list size. We consider the case with ideal SSID list as the reference, representing the case when we can perfectly predict future SSIDs, or when the Wi-Fi radio's memory can store all public SSIDs. This is the best that SSID learning schemes can achieve.

Figure 5.3(a) compares the connectivity achieved by different SSID schemes to that of the ideal SSID list. We also include the algorithm in Android 4.4.2 factory image, which uses the most recently connected SSIDs and adaptive scan interval, referred to as *History-only*. We make three key observations. *First*, Non-uniform consistently reaches 96%+ of the connectivity achieved by the ideal SSID list across all datasets. This verifies the effectiveness of Non-uniform, which incorporates user directionality to better predict future SSIDs. *Second*, compared to the straw-man solutions, Non-uniform achieves larger gain in Keene and Beijing datasets, where hotspots are denser and unevenly distributed. This makes user directionality a critical factor to consider for SSID selection, and enlarges the performance gap between Non-uniform and solutions that do not consider user directionality. *Third*, using purely historical SSIDs leads to poor degradation of Wi-Fi connectivity, especially for Keene and Beijing datasets where users do not have repeated routes and his-



(a) Connectivity (b) Energy

Figure 5.4: Impact of SSID list size on achieved network connectivity and energy. It works under the memory constraint of existing Wi-Fi radio.

torical SSIDs are not relevant. This emphasizes the need of picking SSIDs online based on current location.

Map Information and Historical SSIDs. We further dive into specific design decisions made for Non-uniform, and evaluate their impact on final performance. Specifically, we focus on two design decisions: 1) incorporating map/road information to calibrate the estimation of moving direction, and 2) including historical SSIDs to construct the SSID list. We examine their impact by examining Non-uniform without map information, and Non-uniform without historical SSIDs. Figure 5.3(b) compares their achieved connectivity to that of the original Non-uniform. In comparison, including map information leads to larger gain (10%+) in connectivity than historical SSIDs. This is because street information always helps eliminates directions where feasible paths do not exist if user walk on roads and streets, while historical SSIDs are only helpful for users with regular mobility patterns (*e.g.* the West Lebanon and Canberra datasets). Overall, our results confirm that map information and historical SSIDs are both valuable for Non-uniform to adapt to diverse network setups and user mobility patterns.

SSID List Size. We also examine the impact of SSID list size. In Figure 5.4, we vary

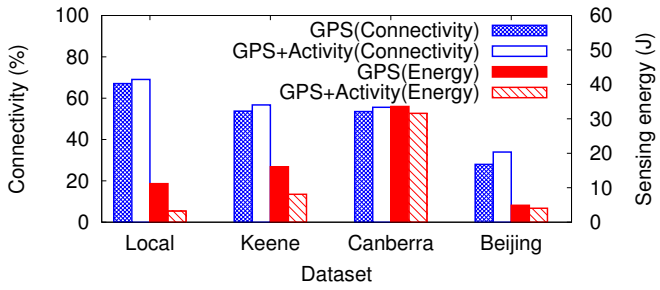


Figure 5.5: Impact of sensing on connectivity and sensing overhead. While adding sensors increases the energy cost for computing SSID list and scan parameters each time, it reduces overall sensing energy by cutting the number of updates on SSID list and scan parameters.

the SSID list size from 4 to 60, and calculate the achieved connectivity using Non-uniform. As our key observation, for all datasets in 5.4(a), the connectivity quickly converges to that of the ideal list once we can offload 16 SSIDs, which is also the maximal number of SSIDs that existing Wi-Fi radio can store. This indicates the efficacy of Non-uniform for predicting the next SSID that the user will encounter. It works under the memory slot constraint of existing Wi-Fi radio. Similarly, for energy curves in 5.4(b), even with 16 SSIDs limitation, Non-uniform method could minimize energy cost in most of cases.

5.4 Sensing Overhead and Performance Gain

Finally, we examine the gain of adding GPS and activity sensors in WiScan, aiming to understand whether the performance gain justifies the sensing overhead. Figure 5.5 shows the achieved network connectivity and total sensing energy cost when WiScan uses only GPS sensor, and when WiScan uses both GPS and activity sensors. The output of activity sensors is used to calibrate the velocity estimation when calculating scan interval. We see that adding activity sensors leads to marginal improvement in achieved connectivity. Surprisingly, the total energy cost on sensing is lower across all datasets. This is because

adding activity sensors fine tunes the calibration of scan interval and thus the timeout value. This reduces the times when offloaded scans hit the timeout value. As a result, adding activity sensors lead to fewer updates of SSID list and scan parameters, and thus lower energy on sensing. For West Lebanon and Keene datasets, adding activity sensors achieves much higher energy saving: it reduces the sensing energy by 70%+ and 50%, respectively. This is because users in these two datasets are non-stationary for a much higher percentage of time (Table 5.1). Hence the use of activity sensors kicks in, and the calibrating of speed estimation leads to higher gain in energy saving for these two datasets. In summary, across all different hotspot distribution and user mobility patterns, adding sensors is beneficial, leading to higher connectivity while reducing the sensing energy for WiScan.

Chapter 6

Related Work

Protocol Offloading. Prior work has offloaded partial upper-layer network protocol to hardware for either performance boost (*13, 16*) or energy saving (*15*). The offloaded protocols range from TCP/IP stack (*13, 16*), to ARP and ICMP (*15*) protocols. Somniloquy (*13*) introduces an always-on, low power embedded secondary processor to augment the PC network interface, and the processor runs an embedded OS networking stack, network port filters and lightweight versions of certain applications (stubs) to continually fetch network request so that it could keep host processor into sleep mode in the meantime to decreases overall energy consumption of PC. Since they need a secondary processor to do these offloading tasks, the system modification would not be neglected. Our work is inspired by these studies, but differs in that we leverage the existing microprocessor on the Wi-Fi radio, and do not introduce a secondary processor. We explore offloading a Wi-Fi protocol task (scan) to radio microprocessor to seek maximal Wi-Fi connectivity with low power.

Low-Power Wi-Fi. There have been active efforts (*18, 19, 22, 24, 29, 30, 32*) on Wi-Fi energy efficiency in the connected state. Catnap (*18*) saves energy by combining tiny

gaps between packets into meaningful sleep interval so that it would allow mobile clients to sleep during data transfers. They introduce a proxy between client and server to analysis and manage the traffic between them so that the multiple transfers would be batch together and mobile device will have more time to sleep. Garcia, et al. conducted a thorough measurement analysis of the power consumption of 802.11 devices that provides a detailed anatomy of the per-packet consumption and characterizes the total consumption of the device (*19*). μPM (*22*) creates sub-states for the Idle state within the 802.11 so that it could determine which sub-state to put the interface into and for how long so that the overall energy consumption is minimized. Sleepwell (*24*) APs adjust client activity cycles to minimally overlap others so that resulting in energy gains with negligible latency on Internet traffic. Rahmati et al. claims a context aware wireless interfaces selection algorithm to leverage the complementary strength of Wi-Fi and cellular networks in order in decrease energy cost (*29*). The most recent design E-MiLi (*32*) reduces the power consumption of Wi-Fi idle-listening by adjusting the radio clock rate.

Our work complements them by focusing on the disconnected state where considerable energy saving are possible. The most related work (*20*) has a mobility-aware system to predict and analysis user mobility pattern and use this pattern to further select best network interface to minimize energy consumption. They use Markov model for Wi-Fi availability prediction, and this will require tons of historical data to obtain a reasonable prediction performance; however, these historical data is not essential for our system. Furthermore, they overlook the energy associated with the main processor during scans. We consider offloading Wi-Fi scans to address this fundamental architectural cause of scan energy inefficiency.

Wi-Fi Connectivity Prediction. Existing work (*17, 26, 27*) on predicting Wi-Fi con-

nectivity provides valuable insights for our algorithmic designs. Deshpande et al. predict Wi-Fi connectivity by using historical driving data and Wi-Fi data along user's driving routes (*17*). BreadCrumbs (*27*) uses Markov Chain to predict Wi-Fi availability and user mobility. Yet they overlook the energy associated with the main processor, work for users with regular mobility pattern, and require periodical/frequent location sensing to achieve accurate prediction. In particular, all of them assume user follows similar route routine. They predict either the AP with the best quality (*26*), or the AP to be encountered in the future (*17, 27*). Furthermore, running as applications, prior works require the main processor to be always active. In contrast, our goal is to achieve energy efficiency while maximizing Wi-Fi connectivity. Our design executes prediction algorithms only when the main processor is active. It computes the SSID list and scan-related parameters online while minimizing the sensing overhead.

Chapter 7

Conclusion and Future Work

The ability of seeking maximal network connectivity is critical to enable future applications that requires tight integration between smart devices and the cloud. Our work aims to push the state-of-the-art further towards low-power ubiquitous network connectivity. We identified and systematically measured the Wi-Fi scan tax problem, the energy inefficiency of existing Wi-Fi scan implementation. Our experiments show that the existing scan method could not find a good balance between saving energy and hunting connectivity because main processor continually consumes energy during the whole Wi-Fi scan period and scan interval could not be automatically adjusted by users' content. In terms of our measurements, by using default scan interval settings within Android framework, the connectivity of users will drop about 80%.

We presented WiScan, a complete system that cuts the scan tax by offloading scan operations to Wi-Fi radio and keep main processor of smartphone sleep during the whole scan tasks. WiScan tackles the challenges in two key design components (computing SSID list, and scan-related parameters) to fully exploit the gain of Wi-Fi scan offloading. SSID list is used to filter unavailable APs out from scan results, and scan-related parameters, namely

scan interval and scan timeout, are used to tell main processor when to update a new list to the radio. We built a WiScan prototype on Google Nexus 5 Android framework and kernel layer. And we evaluated WiScan using both prototype experiments and large-scale emulations driven by real traces with diverse network settings and user mobility patterns from 4 different place around the world. Our results demonstrated WiScan's ability to enable ultra-low power hunting for high Wi-Fi connectivity, which is critical for future context-aware apps that require tight integration between smart devices and the cloud. Both our prototype experiments and trace-driven emulations demonstrate that WiScan achieves at least 90% of the maximal connectivity, while saving 50-62% energy for seeking connectivity compared to existing active scan implementations.

In this study, we have assumed perfect knowledge of AP locations and we assume that open APs are automatically accessible. We plan to study the impact of inaccurate or incomplete hotspot database on WiScan performance and possible design enhancement to make the system more robust. Another design we are investigating is an increase in the Wi-Fi chipset's memory. Although a 10KB increase in Flash RAM significantly increases the bill of materials cost, our scanning algorithms can ensure more pervasive connectivity, and further reduce the Wi-Fi scan tax. WiScan echoes the industrial trend of offloading sensing to low-power co-processors. M7 in Apple iPhone 5s and two co-processors in Moto X has shown the effort of industry to give low-power sensing methods. This trend will give us more freedom to offload and control Wi-Fi tasks based on user context. Finally, while our study has been focusing on Android platform because of the availability of source code of Android framework and open kernel, we plan to examine WiScan on diverse mobile platforms like iOS and Windows Phone and smart devices, especially wearable devices with tight energy budget.

Bibliography

1. https://republicwireless.com/.
2. http://www.freedompop.com/.
3. http://www.qca.qualcomm.com/technology/technology.php?nav1=47&product=67.
4. http://www.marvell.com/wireless/avastar/88W8787/.
5. http://www.technologyreview.com/news/517801/the-era-of-ubiquitous-listening-dawns/.
6. http://www.technologyreview.com/news/519531/what-apples-m7-motion-sensing-chip-could-do/.
7. http://www.msoon.com/.
8. http://www.ifixit.com/Teardown/.
9. http://crawdad.cs.dartmouth.edu/.
10. http://developer.android.com/reference/com/google/android/gms/location/ActivityRecognitionClient.html.
11. http://www.jiwire.com/.

12. https://developers.google.com/maps/documentation/geocoding/.

13. AGARWAL, Y., ET AL. Somniloquy: augmenting network interfaces to reduce PC energy usage. In *Proc. of NSDI* (2009).

14. BERTRAM, J. E. A., ET AL. Multiple walking speedfrequency relations are predicted by constrained optimization. *Journal of Theoretical Biology* (2001), 445–453.

15. CHRISTENSEN, K. J., ET AL. The next frontier for communications networks: power management. *Computer Communications 27*, 18 (2004), 1758–1770.

16. CURRID, A. TCP offload to the rescue. *Queue 2* (May 2004), 58–65.

17. DESHPANDE, P., ET AL. Predictive methods for improved vehicular WiFi access. In *Proc. of MobiSys* (2009).

18. DOGAR, F. R., STEENKISTE, P., AND PAPAGIANNAKI, K. Catnap: exploiting high bandwidth wireless interfaces to save energy for mobile devices. In *Proc. of MobiSys* (2010).

19. GARCIA-SAAVEDRA, A., ET AL. Energy consumption anatomy of 802.11 devices and its implication on modeling and design. In *Proc. of CoNEXT* (2012).

20. KIM, K.-H., ET AL. Improving energy efficiency of Wi-Fi sensing on smartphones. In *Proc. of INFOCOM* (2011).

21. LIU, J., ET AL. Energy efficient GPS sensing with cloud offloading. In *Proc. of SenSys* (2012).

22. LIU, J., AND ZHONG, L. Micro power management of active 802.11 interfaces. In *Proc. of MobiSys* (2008).

23. LONG, L. L., AND SRINIVASAN, M. Walking, running, and resting under time, distance, and average speed constraints: optimality of walkrunrest mixtures. *Journal of The Royal Society Interface 10*, 81 (2013).

24. MANWEILER, J., AND ROY CHOUDHURY, R. Avoiding the rush hours: WiFi energy management via traffic isolation. In *Proc. of MobiSys* (2011).

25. MEGIDDO, N., AND MODHA, D. ARC: A self-tuning, low overhead replacement cache. In *Proc. of FAST* (2003).

26. NICHOLSON, A. J., CHAWATHE, Y., CHEN, M. Y., NOBLE, B. D., AND WETHERALL, D. Improved access point selection. In *Proc. of MobiSys* (2006).

27. NICHOLSON, A. J., AND NOBLE, B. D. BreadCrumbs: forecasting mobile connectivity. In *Proc. of MobiCom* (2008).

28. RA, M.-R., ET AL. Improving energy efficiency of personal sensing applications with heterogeneous multi-processors. In *Proc. of UbiComp* (2012).

29. RAHMATI, A., AND ZHONG, L. Context-for-wireless: context-sensitive energy-efficient wireless data transfer. In *Proc. of MobiSys* (2007).

30. ROZNER, E., ET AL. NAPman: network-assisted power management for WiFi devices. In *Proc. of MobiSys* (2010).

31. SCELLATO, S., ET AL. NextPlace: A spatio-temporal prediction framework for pervasive systems. In *In Proc. of Pervasive* (2011).

32. ZHANG, X., AND SHIN, K. G. E-MiLi: energy-minimizing idle listening in wireless networks. In *Proc. of MobiCom* (2011).