Low-Power Pervasive Wi-Fi Connectivity Using WiScan

Tianxing Li, Chuankai An, †Ranveer Chandra, Andrew T. Campbell, and Xia Zhou
Department of Computer Science, Dartmouth College †Microsoft Research
{ltx, chuankai, campbell, xia}@cs.dartmouth.edu ranveer@microsoft.com

ABSTRACT
Pervasive Wi-Fi connectivity is attractive for users in places not covered by cellular services (e.g., when traveling abroad). However, the power drain of frequent Wi-Fi scans undermines the device's battery life, preventing users from staying always connected and fetching synced emails and instant message notifications (e.g., WhatsApp). We study the energy overhead of scan and roaming in detail and refer to it as the *scan tax problem*. Our findings show that the main processor is the primary culprit of the energy overhead. We propose a simple and effective architectural change of offloading scans to the Wi-Fi radio. We design and build *WiScan* to fully exploit the gain of scan offloading. Our experiments demonstrate that WiScan achieves 90%+ of the maximal connectivity, while saving 50-62% energy for seeking connectivity.

Author Keywords
Wi-Fi connectivity; Wi-Fi scans; energy efficiency

ACM Classification Keywords
C.2.1 Computer-Communication Networks: Network Architecture and Design

INTRODUCTION
Wi-Fi operates over unlicensed spectrum and is easy to use. Hence, it is often a preferred, low-cost alternative to connect to the Internet. Businesses, such as coffee shops, restaurants, and city municipalities, frequently offer free (or inexpensive) Wi-Fi connectivity in exchange for user's data and time on their premises. In fact, large portions of big cities are blanketed by Wi-Fi hotspots [13, 21, 23, 27, 28], such that a user can be connected to the Internet 24/7 [13].

Several applications would benefit if a mobile device could continuously stay connected over Wi-Fi. For example, when a user is traveling internationally and the cellular data plans are expensive. The users' mobile device could stay in sync with his/her e-mail, WhatsApp, and Skype messages. In another example, in countries where users have pay-per-byte plans, the monthly bill could be reduced by maximizing connectivity over Wi-Fi, especially for traffic-heavy applications on file synchronization (e.g., Dropbox, Google Drive) or music streaming (e.g., Pandora, Spotify). Furthermore, this ability could give rise to new business models (e.g., an inexpensive, Wi-Fi only Skype or Hangouts phone [8, 20]). It also serves as an appealing means to complement existing cellular services. Recent Microsoft WiFi service [15] and Google's Project Fi [18] both leverage millions of Wi-Fi hotspots worldwide to provide pervasive connectivity.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
UbiComp '15, September 7–11, 2015, Osaka, Japan.
Copyright 2015 © ACM 978-1-4503-3574-4/15/09...$15.00.
http://dx.doi.org/10.1145/2750858.2807515

However, achieving pervasive Wi-Fi connectivity is nontrivial. The biggest challenge is the power consumed by frequent Wi-Fi scans. Unlike cellular modems, the Wi-Fi chipset and the main processor need to stay awake to scan for available Wi-Fi Access Points (APs) and roam across small Wi-Fi cells. In our experiments, scans from Skype (every 60 s) reduced the battery life of a Nexus 4 by 43%. Even the default 300 s scan interval reduces the battery life by 19%.

As we investigate the energy overhead of Wi-Fi scans, we find that the primary culprit is the main processor, which has to be active during the scan operation. From our measurements, the main processor typically consumes 1-2 times more energy than the Wi-Fi radio during the scan. We refer to the main processor's energy cost as the *scan tax* paid for each scan operation. As increasing applications rely on aggressive scanning for better Wi-Fi connectivity, it is critical to find solutions to the scan tax problem. Existing solutions either turn off the Wi-Fi radio, or rely on users' manual hunting for Wi-Fi hotspots, both leading to a poor user experience.

In this paper, we consider a simple architectural change that can significantly drive down the scan tax by offloading the Wi-Fi scan to Wi-Fi radios. The key idea is as follows. *First*, the main processor computes a list of SSIDs and scan-related parameters (i.e., scan interval, timeout) by considering AP locations and user mobility with the goal of maximizing Wi-Fi connectivity. For secure SSIDs that require authentication, the login passwords are also offloaded to the Wi-Fi radio. Using techniques such as Hotspot 2.0 (or Wi-Fi Passpoint) [24, 59] and WISPr [34], we can automate the authentication without requiring user interactions. *Second*, the Wi-Fi radio performs periodic scans and wakes up the main processor only when it discovers any SSID in the SSID list. The main processor then associates with the discovered SSID. By reducing the active duration of the main processor, offloading scan improves the energy efficiency of the disconnected state. In fact, part of this architectural change has emerged in recent products [6, 17]. However, the current adoption uses a very rudimentary design, which can lead to significant (up to 90%) connectivity loss and is unable to support various use cases (e.g., traveling). So far, little is known on the full potential of scan offloading and how to effectively unleash it.

We aim to bridge this gap in this paper. To fully exploit the benefits of scan offloading, we face three key challenges.

Table 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 1.4 GHz	TI OMAP 4460 dual-core 1.2 GHz	Snapdragon S4 Pro quad-core 1.5 GHz	Snapdragon 800 quad-core 2.26 GHz	Exynos 5 quad-core 1.9 GHz	TI OMAP 4430 dual-core 1.2 GHz
Battery	2100 mAh, 3.8 V	1750 mAh, 3.7 V	2100 mAh, 3.8 V	2300 mAh, 3.8 V	3200 mAh, 3.8 V	570 mAh, 3.7 V
Baseline power	8.87 mW	18.31 mW	14.04 mW	12.24 mW	12.70 mW	23.87 mW

First, computing the offloading SSID list is nontrivial. Wi-Fi chipsets have very limited memory and can store only up to 10-16 SSIDs [19, 14], while users typically encounter 2-3 times more SSIDs even in half an hour [1, 2]. A larger on-chip memory mitigates the problem and yet entails a higher material cost, unlikely to be adopted by hardware vendors since Wi-Fi chipset is a low-end market. *Second*, user mobility complicates the configuration of scan parameters, which depends on the user's location, moving speed/direction, and the density of surrounding APs. *Third*, smart devices have limited computational power and battery. We need low-complexity algorithms to adapt the SSID list and scan parameters online without much additional sensing overhead. In the future, the adaptation process may be fully offloaded to the Wi-Fi radio as the mobile architecture advances [55].

To address the above challenges, we design and build *WiScan*. To compute the offloaded SSIDs, WiScan uses a lightweight learning scheme that integrates historical SSIDs and user mobility prediction to estimate the SSIDs that the user will encounter in the near future. Unlike prior work that typically requires frequent sensing (e.g., location, accelerometer) [43, 51, 58], our SSID learning scheme does not require continuous sensing and makes no assumption on AP's coverage. To configure scan parameters, WiScan integrates the output of low-power activity sensors 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. Our results demonstrate WiScan's energy efficiency across diverse network deployments and user mobility patterns. It achieves 90%+ of the optimal Wi-Fi connectivity, yet reduces 50-62% of the energy on seeking Wi-Fi connectivity.

Our key contributions are as follows:

- We analyze the Wi-Fi scan tax problem, conduct measurements on diverse smart devices to examine the root cause, and quantify its impact on device's battery life;
- To cut the Wi-Fi scan tax, we design intelligent algorithms that compute the offloading SSID list and scan-related parameters to fully exploit the benefits of scan offloading;
- We build a WiScan prototype on the Nexus 5 and perform real-world experiments to validate WiScan's significant energy saving and near-optimal connectivity;
- We evaluate WiScan using large-scale emulations driven by scan traces of smartphone users across the world.

We believe that WiScan represents a significant departure from prior efforts [33, 43, 50, 60, 64] by tackling the root cause of Wi-Fi scan energy inefficiency. The principle of WiScan can potentially be extended to other types of radios (e.g., cellular, white spaces radios). WiScan is similar in spirit to the industrial trend (e.g., Apple's M7, Moto X) of offloading sensing to low-power co-processors [25, 26]. Ultimately, WiScan can enable "always-on" Wi-Fi connectivity necessary for future context-aware applications such as Glass-based augmented reality and gesture-driven HCI.

WI-FI SCAN TAX PROBLEM

We first examine the problem of Wi-Fi scan tax and analyze its implications on maximizing Wi-Fi connectivity.

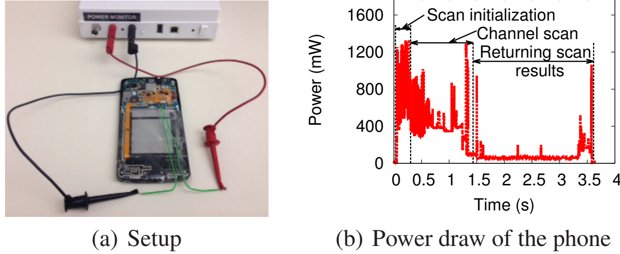
Energy Consumption of Wi-Fi Scan

To examine the energy consumed by Wi-Fi scans, we test five models of Android smartphones and Google Glass (Table 1). We measure the device's power draw using the Monsoon power monitor [16], which reports instantaneous power averaged every 0.2 ms. As shown in Figure 1(a), we remove the device battery, and connect battery pins to the power monitor[1]. 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 scans with a specified interval without associating with any AP. To obtain clean 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 1) align with prior results [11].

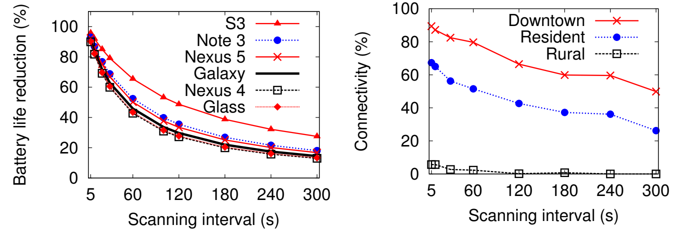
For all measured devices, we observe that their Wi-Fi radios perform active scans when the screen is off[2]. Figure 1(b) plots the total power draw of Nexus 5. The scan consists of three phases: scan initialization (0 - 0.3 s), channel scan (0.3 - 1.4 s), and returning scan results to the application layer (1.4 - 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 0.74 J energy. We observe similar patterns when measuring other devices (Table 2).

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

[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.



(a) Setup (b) Power draw of the phone



(c) Impact on battery life

Figure 1. Energy measurements of Wi-Fi scans using Nexus 5. We connect a Monsoon power meter to the phone's battery pins (a). (b) shows the total power consumption of the phone during a scan, and (c) shows that frequent Wi-Fi active scans significantly reduce the battery life.

Figure 2. Impact of Wi-Fi scans on Wi-Fi connectivity, using a dataset collected in Keene, NH. Default scan intervals (300 s or 240 s) reduce Wi-Fi connectivity by nearly half.

The energy overhead of Wi-Fi scan significantly affects the device's battery life. To examine this impact, we vary Wi-Fi radio's scan interval from 5 s (the minimal) to 300 s, and compare the resulting battery life B^{T_s} under scan interval T_s to the baseline battery life B when the device is in the disconnected idle state consuming the baseline power. We derive B^{T_s} and B as follows. Assume the device has a battery with V volts and A mAh, its baseline power is P_b in mW, and the energy consumed by a Wi-Fi scan is E_s in mJ (Table 2, not including baseline power), then we have $B = (A \cdot V)/P_b$ and $B^{T_s} = (A \cdot V)/(E_s/T_s + P_b)$. Figure 1(c) shows the battery life reduction $(1 - B/B^{T_s})$ of different devices under each scan interval T_s. We see that across all devices, frequent Wi-Fi scans significantly drain the battery, reducing the battery life by up to 90%+. 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., 300 s for Android framework). Next, we examine the impact of scan interval on the resulting Wi-Fi connectivity.

Impact on Wi-Fi Connectivity

We built a system service to collect Wi-Fi scan traces and user mobility. Unlike prior Wi-Fi traces [7] that are collected using default Wi-Fi scan intervals (180 s or 300 s), 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 received by the user. Specifically, our service forces the Wi-Fi radio to scan every 5 s 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 [9]. We have collected Wi-Fi scan traces from four cities across the world: our local city, Keene in New Hampshire, Canberra in Australia, and Beijing in China, with 4 users over 7 days. Table 3 summarizes the dataset statistics.

Using these Wi-Fi scan traces as ground truth, we now examine the resulting Wi-Fi connectivity using different scan intervals. We assume the device can only connect to public SSIDs, and it connects to the SSID with the strongest received signal until it no longer sees this SSID during the scan. For a given scan interval, we calculate the user's connectivity as the percentage of time that the user connects to any SSID.

Take the dataset at Keene as an example. In this 5 hr walking trace, the user passes traffic lights, parking lots, buildings, and bridges. Figure 2 plots the connectivity as the scan interval varies from 5 s to 300 s. We identify three types of areas: downtown area (120 APs per km^2), residential area (40 APs per km^2), and rural area (5 APs per km^2). We make two key observations. *First,* across all different areas, scan frequency significantly affects 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 Android framework) reduces Wi-Fi connectivity by at least half. *Second*, in areas with relatively sparser APs (i.e., residential and rural areas), scanning frequently is essential to seize the sparse connectivity. Overall, maximizing Wi-Fi connectivity needs frequent Wi-Fi scans, which however drain the battery significantly (Figure 1(c)). Thus, the energy inefficiency of scan is a big hurdle for maximizing Wi-Fi connectivity.

Wi-Fi Scan Tax

We quantify the scan tax and analyze its associated activities.

Quantifying the Wi-Fi Scan Tax

We quantify the scan tax by measuring the main processor's power during an active scan. This is challenging because individual components (e.g., Wi-Fi radio, main processor) do not expose pins for us to measure their power draw. Although the Android framework provides an estimated energy breakdown for each component, it uses the built-in battery sensor far less accurate than 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 verified that the ghost service consumes negligible energy, so the power monitor readings accurately reflect main processor's power draw.

Table 2 shows the energy breakdown and the active duration of the Wi-Fi radio (and the bus) and main processor during an active scan. 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 3–4 s,

Table 2. Energy consumption and active duration of the Wi-Fi radio and the main processor (MP) for an active Wi-Fi scan. We do not include the baseline power in MP's energy numbers to truly reflect MP's energy consumption associated with the scan. Across all smart devices, MP's energy consumption consistently occupies nearly 60% of the total energy of a Wi-Fi scan.

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

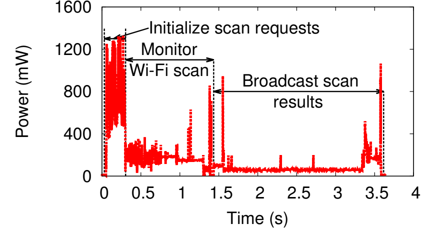
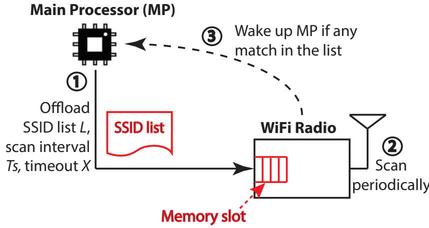
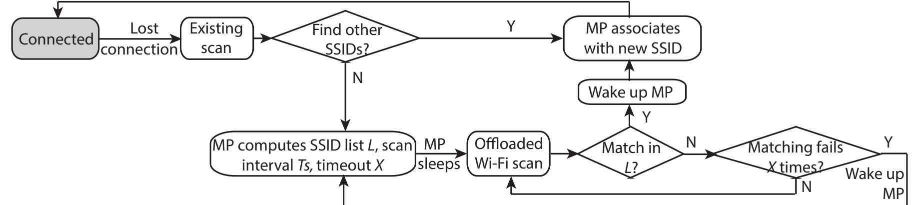


Figure 3. The Wi-Fi scan tax: main processor's power draw in a Wi-Fi active scan (Nexus 5).



(a) Offloading Wi-Fi scan



(b) Flowchart of Wi-Fi offloading

Figure 4. 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.

much longer than the radio. This high scan tax fundamentally limits the potential of maximizing Wi-Fi connectivity.

How is the Wi-Fi Scan Tax Spent?

We further analyze the main processor's activities during a Wi-Fi scan. As shown in Figure 3, the main processor performs the following operations:

- *Initializing scan requests.* The main processor checks Wi-Fi radio status (e.g., connection state, supplicant pending state, and driver state). If it is valid, the main processor initializes an I/O buffer to store the package from Wi-Fi radio. It then invokes the Wi-Fi driver to prepare a scan request that includes the scan type, scan interval, and scan timeout. The processor sends the scan request to Wi-Fi firmware. The peak power of these operations is 1300 mW.
- *Monitoring Wi-Fi scan.* The main processor monitors the scan result as the radio sends a scan probe to each channel sequentially. Once the radio finishes scanning a channel, the main processor collects the scan result and counts APs on this 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 app requests. If no request comes, the processor releases the I/O buffer and sets the timer for the next scan, leading to a peak power of 1000 mW.

WiScan: OFFLOADING WI-FI SCAN

We present WiScan to cut the scan tax. We describe the concept of offloading scans and the detailed design of WiScan.

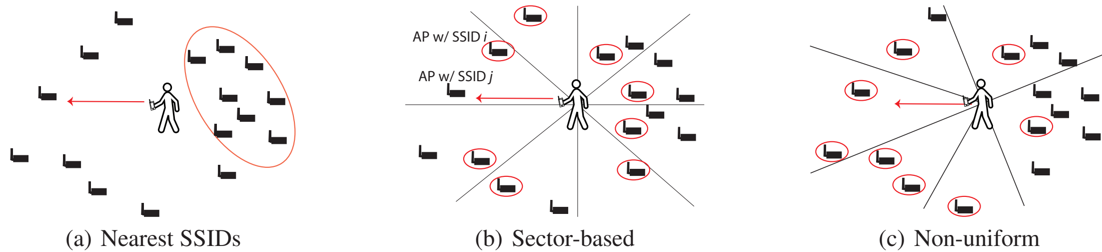
Concept and Design Challenges

Figure 4(a) illustrates the concept. The device offloads the scan when it is not connecting to any AP and the screen is off. Before entering the sleep mode, the main processor computes a list of SSIDs and scan-related parameters (scan interval and 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 periodically scans and compares its discovered SSIDs to the SSID list. If any match is found, the radio wakes up the main processor to associate with the matched SSID[3]. If there are multiple SSID matches, it picks the SSID with the strongest signal. With the SSID list, we can 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, avoiding waking up the main processor for APs unable to connect to (e.g., unsubscribed APs). The radio also wakes up the processor if it cannot find any matches after X independent scans, so that the system is not stuck with outdated SSID list and scan parameters (Figure 4(b)).

The concept of scan offloading can be generalized in two aspects. First, we can consider offloading BSSIDs. As unique identifiers of wireless routers, BSSIDs are much harder to manipulate than SSIDs and thus offer better security. Second, we can design more sophisticated metric to select SSIDs/BSSIDs, which can take into account not only signal strength but also its bandwidth and other performance metrics. We leave these discussions to future work.

To realize the concept of offloading Wi-Fi scan, we need to determine: 1) the offloading SSID list so that the device does not miss any available Wi-Fi connectivity; and 2) the scan frequency and timeout to avoid unnecessary scan operations and thus save energy. Configuring these parameters brings three challenges. *First*, off-the-shelf Wi-Fi chipsets have very limited memory, which can only store up to 10-16 SSIDs [19, 14] and is costly to increase. Using our Wi-Fi scan traces

[3]For SSIDs that require login, Hotspot 2.0 [24] and WISPr [34] can be used to automate the login and avoid user interaction.



(a) Nearest SSIDs (b) Sector-based (c) Non-uniform

Figure 5. Schemes of deciding SSID list assuming $N = 8$, with selected SSIDs marked by red circles. (a) 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; (b) Sector-based scheme selects SSID in all directions. But an SSID per sector still can lead to miss connectivity. Here the user will encounter SSID$_j$, rather than the selected SSID$_i$; (c) Non-uniform scheme picks more SSIDs in the estimated user's moving direction, and thus better predicts the SSID the user will encounter.

and other existing traces [1, 2], our analysis shows that a user typically can encounter 2-3 times more SSIDs in even half an hour. So it is nontrivial to decide the SSID list that maximizes the Wi-Fi connectivity. It becomes more challenging when we generalize the idea to offloading BSSIDs. *Second*, user mobility further complicates the configuration. Prior work on mobility prediction [51, 58] requires training using long-term historical data and assumes that users have a regular routine. These assumptions, however, do not always hold (e.g., users travel to new places). We need to predict future SSIDs on the fly even without historical data. *Third*, smart devices have limited energy and computation power. Thus the algorithm has to be low-complexity and energy-efficient. Next we describe our solutions to addressing the above challenges.

Computing the Offloading SSID List

The first key design component is to determine the SSID list to offload to the Wi-Fi radio. The locations of public hotspot can be easily obtained in public databases such as JiWire [12] and wefi [22], which contain over 100 million Wi-Fi hotspot entries around the world. 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 set L to the most popular SSIDs. To examine how well this approach works, we crawled large-scale hotspot data[4] 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 found that popular SSIDs cover only 40%+ of all SSIDs' coverage and solely relying on them leads to significant connectivity loss (>50%).

The above observations turn our attentions to approaches that compute L online. Existing industry approach caches the N most recently connected SSIDs [6, 17], referred to as the *history-only* scheme. While simple, this approach fails when users travel to new places or have irregular mobility patterns. Another intuitive approach is to select the N SSIDs nearest to the user, referred to as the *nearest-SSIDs* scheme. However, by ignoring the user's moving direction, this approach can result in incorrect prediction of the next available SSID. Figure 5(a) illustrates a simple example, where the user's nearest N SSIDs are clustered. The user is walking away from their coverage and can no longer connect to any of them. However, the nearest-SSIDs scheme still predicts these SSIDs, which clearly are no longer relevant to the user.

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 5(b)). This scheme is simple and does not require mobility estimation. But it is limited by the small number of sectors (i.e., the number of SSIDs N), which can still lead to connectivity losses (Figure 5(b)).

We propose 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 in sectors in user's moving direction (Figure 5(c)). 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. Using SSIDs' distances to the user, our method makes no assumption on hotspot coverage area, which is typically irregular due to the complex wireless signal propagation. Our method takes the following input: 1) the user's current location, which can be obtained by using existing localization techniques [35, 37] or the GPS sensor), and 2) map information (e.g., street). Note that to lower the overhead of acquiring user's location, WiScan adapts the GPS sampling rate based on user's current mobility status. We plan to leverage low-power localization techniques [45, 52, 54, 56, 61] to further minimize the overhead. Our approach has three steps.

Step #1: Predicting Direction. We predict user's moving direction if the user is non-stationary based on the activity inference [9]. Assume p_1 is user's current location at time t_1, we estimate user's moving direction as $\overrightarrow{p_0p_1}$, where p_0 is the latest location that is at least 10 m away from p_1 and $t_1 - t_0 > 60$ s. This is to reduce the impact of potential location sensing error (e.g., GPS or Wi-Fi localization error can be higher than 10 m). We further refine the direction estimation using street information to eliminate directions where feasible paths do not exist. Overall, our prediction method does not require frequent location tracking and works upon sparse location data.

[4]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 [10] to obtain hotspot GPS coordinates based on their addresses on Jiwire.

Step #2: Partitioning Sectors. Based on the estimated moving direction $\overrightarrow{p_0p_1}$, we first divide the space around the user into N uniform sectors, such that $\overrightarrow{p_0p_1}$ is the angular bisector of a sector. We classify the N sectors into two groups: forward sectors whose angular bisectors have acute angles to $\overrightarrow{p_0p_1}$, and backward sectors whose angular bisectors have obtuse angels to $\overrightarrow{p_0p_1}$. Then we adjust the sector partition by merging every x adjacent backward sectors and obtain $\lfloor N/(2x) \rfloor$ backward sectors. We exhaustively tested different x values offline, and $x = 2$ performs the best in all experiments. By assigning more sectors in the forward direction, we increase the likelihood of including 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. Figure 5(c) shows a simple example, where 4 backward sectors are merged into 2 sectors, resulting into 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 select the nearest SSID in each sector, which, however, can provide wrong estimation because of the complex wireless propagation. Instead, we consider both SSIDs' distances and the connection history. Our selection works as follows. *First*, we select $2N$ nearest SSIDs as candidate SSIDs. This is similar in spirit to the ghost list design in adaptive caching algorithm (ARC) [49]. For these $2N$ SSIDs, we rank the historical SSIDs by the number of times they have been connected to in a descending order. *Second*, for each sector, we select the top-ranked historical SSID in this sector. If no historical SSID resides in a sector, we resort to the nearest SSID in this sector. We obtain M SSIDs in the end. *Finally*, we select the $(N - M)$ nearest SSIDs from the remaining new SSIDs in forward sectors. This completes the SSID selection.

Configuring Scan Parameters

The second component is to determine the scan interval T_s and timeout X of offloaded Wi-Fi scan. We adapt both parameters based on the nearby hotspot distribution and user's mobility pattern. Unlike prior work [43], our configuration makes no assumption on hotspot coverage shape and size.

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). It works as follows. *First*, from activity inference, we obtain user's current motion status: 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 (1000 s) that the hardware can support, similarly to the existing technique [60]. If the user is non-stationary, we estimate the moving velocity $\vec{v}$ using the current location p_1 and the previous location p_0, where $\vec{v} = |p_1 - p_0|/(t_1 - t_0)$. 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 [36, 48]. If

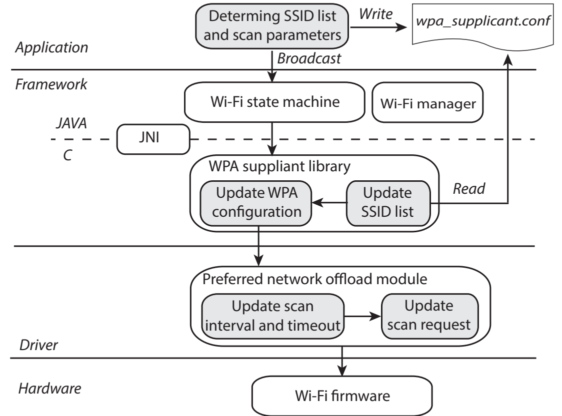


Figure 6. WiScan implementation on Nexus 5 (Android 4.4.2 developer image). Shaded blocks are modified/added modules. SSID list configuration is implemented above the driver level, and scan parameters (scan interval, timeout) are adapted in the driver.

$\vec{v} < v^{min}$, 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: 0–10 s, 10–40 s, and 40–70 s. 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 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 compute X by estimating the duration when the user can maintain connectivity to any SSID in the list. Specifically, based on the estimated moving direction, we derive the probability p_i of the user moving towards sector i assuming p_i follows a standard normal distribution $\mathcal{N}(0, 1)$, where the sector in the estimated direction has the highest probability. So the expected distance $\overline{d}$ to the furthest SSID is $\overline{d} = \sum_{i=1}^{N} d_i * p_i$, where d_i is the distance to the furthest SSID in sector i. The X value is then set as $\overline{d}/d_{min}$, where d_{min} is the distance to the nearest SSID in the list. This approximates the number of scans before the user loses connection to any SSID in the current list.

WiScan PROTOTYPE EVALUATION

We build a proof-of-concept prototype of WiScan and examine its practical performance.

WiScan Implementation

We implement WiScan on the Nexus 5 Android phone (4.4.2 OS developer image). The phone uses the Broadcom BCM4339 Wi-Fi chipset, where the Wi-Fi scan operation is

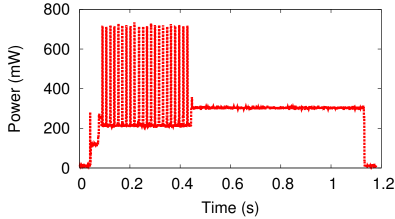
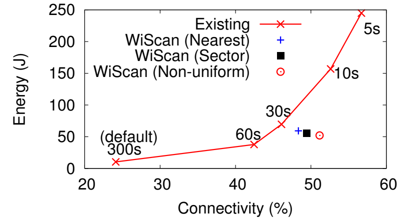
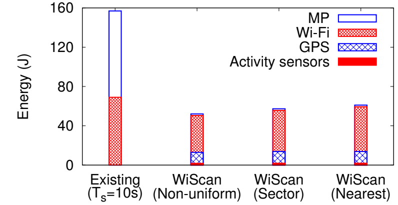


Figure 7. Indoor energy measurement of a Nexus 5 during an offloaded scan in WiScan. After removing the scan tax, a single scan consumes 0.33 J energy, down from 0.74 J of the existing scan (Table 2).



(a) Overall performance



(b) Energy breakdown

Figure 8. WiScan outdoor field experiments at a local city. We carried multiple Nexus 5 phones, where one phone runs WiScan, and the others use existing scans under different scan intervals. We observe that WiScan reduces 67% energy compared to existing scan while achieving similar connectivity.

built in the chipset's firmware, and the chipset has a 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 (15 s)[5] independently, and wakes up the main processor only if 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 enable real-time configuration of the SSID list and scan parameters using proposed algorithms. We accomplish these tasks by modifying and adding related modules at the driver, framework, and application level. Figure 6 shows the system architecture of WiScan. Specifically, we implemented SSID list selection above the driver layer. Our SSID learning scheme is implemented at the application layer as a background system service. It writes an SSID list in the `wpa_supplicant.conf` file, which will be read by our added module in the WPA supplicant library in the 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 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 the Wi-Fi firmware. To avoid the delay (≈1 s) of cleaning cached scan requests, we set up a parallel thread to expedite scan offloading.

Indoor Energy Measurements

We start with examining the energy consumption of an offloaded Wi-Fi scan in WiScan. We follow the setup in Figure 1(a), and plot in Figure 7 the instantaneous power draw of the phone during an offloaded scan. Comparing it to the existing scan (Figure 1(b)), we make the following observations. *First*, by removing main processor's operations, an offloaded scan is much faster (only 1.1 s), while an existing scan lasts for 3.6 s. Specifically, it has two phases: 1) the radio probes 22 channels, generating 22 energy spikes 1.7-2 ms each (700 mW+ stays for 1 ms). The radio compares the discovered SSIDs to the SSID list; and 2) the radio releases the chipset buffer storing scan results and prepares for the next scan, resulting into 300 mW+ power draw. *Second*, the offloaded scan consumes 56% less energy by cutting the scan tax. The scan energy comes from only the Wi-Fi radio (0.32 J) plus the phone's baseline energy (0.01 J in Table 2). We expect similar energy reduction (60%+) for other devices.

Next we examine the energy overhead of computing SSID list and scan parameters in WiScan. The overhead is from: 1) the main processor computing SSID list and scan parameters, 2) the location sensor, and 3) the activity sensors for Google activity inference. We measure the energy of each component by instrumenting the phone to perform each operation. We observe that the main processor consumes 0.1 J to run our algorithms[6], GPS on Nexus 5 costs 0.7 J to return a location, and activity sensors cost 0.1 J to infer activity. Note that all the overhead is shared by multiple offloaded scans, since the system only updates the SSID list and scan parameters when it first enters the mode of scan offloading or when offloaded scans fail to find any SSID matches after hitting the timeout.

Outdoor Field Experiments

Next we conduct WiScan experiments in a local shopping area. The area is roughly 4 km^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 SSID information is saved on the phone. In the experiments, users walk casually (0.2 – 2 m/s) with occasional pauses while carrying three Nexus 5 phones. One phone is implemented with WiScan and the others use existing scan. We log the active duration of GPS, activity sensors, main processor, and Wi-Fi radio. We derive the energy consumption of these sensors using offline energy measurement. We repeat the experiment in three rounds and test existing scan with a fixed scan interval from 5 s (the minimal) to 300 s (the default). Each round lasts 2 hours with the same walking route.

We focus on Wi-Fi connectivity (the percentage of connected time) and the energy cost of seeking Wi-Fi connectivity. Figure 8(a) compares existing scans to WiScan with three SSID algorithms. Since WiScan automatically adapts its scan interval based on user's current location and mobility, its performance is shown as a point in the figure. In comparison,

[5]In Android 4.4.2 factory image, the scan interval starts from 15 s, and then doubles after 4 failed tries. It is capped by 240 s.

[6]We have measured the energy consumption of different SSID selection algorithms, and observed the same number.

Table 3. Summary of four Wi-Fi scan traces, collected from smartphone users at four cities across the world.

Dataset (City)	# of public SSIDs	Area (km^2)	Duration	Stationary duration	Setting
Local	44 (75 APs)	4	10.5 hr	19%	Outdoor
Keene	148 (183 APs)	6	5 hr	27%	Outdoor
Canberra	72 (241 APs)	9	21.8 hr	88%	Indoor & Outdoor
Beijing	41 (41 APs)	2	4 hr	63%	Outdoor

existing scans use a fixed scan interval without adaptation, so we manually vary the scan interval and plot the results as a line. Clearly a shorter scan interval leads to higher connectivity, as the radio is seeking for hotspots more aggressively and hence is less likely to miss available connectivity. Compared to existing scans, WiScan significantly reduces the energy cost while achieving similar connectivity. Compared to existing scans with the highest frequency ($T_s = 5$ s), WiScan achieves 90% of its connectivity using only 21% of its energy. WiScan achieves similar connectivity to existing scan with $T_s = 10$ s, yet reduces the energy by two thirds.

Figure 8(b) further shows the energy breakdown of WiScan and the existing scan with $T_s = 10$ s. We make three key observations. *First*, WiScan's energy saving is contributed by two factors. The main factor is cutting the scan tax, where the main processor costs 56% of the energy for existing scan, yet only 3% in WiScan. Another factor is adapting the scan interval and timeout, which avoids unnecessary scans when no hotspots are nearby or the user is stationary. *Second*, WiScan brings additional sensing overhead, dominated by GPS (22% of overall energy). The GPS cost can be driven down with the recent low-power GPS design [46]. We can also leverage other localization techniques [35, 37] to obtain user location without turning on GPS. *Third*, among SSID learning schemes, Non-uniform moderately outperforms others. By leveraging user's moving direction, it predicts future SSIDs more accurately, leading to 4-6% increase in connectivity.

The energy saving of WiScan greatly increases the phone's battery life. In our experiments, all phones are fully charged initially. We switch off phone screens and disable all radios except Wi-Fi. After each round of the experiment, the phone using WiScan consumes 6% of the battery life, while other phones consumes 17% of the battery life to achieve the same Wi-Fi connectivity.

TRACE-DRIVEN EMULATIONS

After examining the WiScan prototype, we now use large-scale emulations to examine WiScan in diverse network deployments and the impact of design choices in WiScan.

Emulation Setup

We developed an emulator using Python to examine WiScan and existing scan implementation. The emulator takes our collected Wi-Fi scan traces (Table 3) as the ground truth of Wi-Fi connectivity at each location. We obtain hotspot locations using public hotspot databases (JiWire and wefi) and manual labeling. The users in our traces consist of office workers, students, and professors. Their activities include sitting still, walking (0.5 – 2 m/s), biking, and driving (3 – 15 m/s). Table 3 summarizes dataset statistics.

We assume that a device can connect to an SSID if the device discovers this SSID and its received signal strength is above -90 dBm, which is the signal threshold observed in our experiments. We emulate WiScan based on Figure 4(b). We consider the existing scan implementation as the baseline, referred to as *Existing*, and examine Existing with the fixed scan interval from the minimal (5 s) to the default (300 s). We also compare WiScan to a prior mechanism WiFisense [43], which leverages periodical activity sensing to adapt scan interval. We configure WiFisense's parameters so that they best fit our datasets and examine WiFisense with activity sensing frequency (f) from 5 s to 300 s.

To evaluate the above mechanisms, we focus on their achieved connectivity and energy cost. We define connectivity as the percentage of connected time. From our experiments, we observe that AP association takes 4 s and obtaining GPS location takes 3 s. So we subtract these delays from the connected time. We define energy cost as the energy consumed in the disconnected state, which is computed using our measurement numbers (Table 2).

Overall Performance

To evaluate the overall performance of WiScan, Existing, and WiFisense, we compute 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 connectivity. Figure 9 plots the connectivity-energy tradeoff and Figure 10 shows the energy breakdown when these mechanism achieve similar connectivity. There is only a single point for WiScan in Figure 9 because WiScan automatically adapts it scan interval based on user mobility and AP distribution. However, WiFisense and existing method in Android work with a fixed scan interval, so we configure their fixed scan interval to different values and examine their tradeoff between energy consumption and achieved connectivity. We also include WiScan without the automatic adaptation of scan interval, referred to as *WiScan (Fixed)*, with the goal of understanding the contribution of adapting scan intervals. Our key observations are as follows.

Observation 1: WiScan achieves 90%+ optimal connectivity. Across all four datasets, WiScan consistently achieves 90%+ of the optimal connectivity, demonstrating that our SSID learning scheme accurately predicts future SSIDs. In particular, WiScan achieves 97% of the optimal in the Canberra dataset. This is because the user in this dataset is mostly stationary (Table 3) and follows a regular mobility pattern (work and home), which eases the SSID prediction. We notice that although the Beijing dataset has a dense hotspot deployment (20 per km^2), its connectivity percentage is lower than other datasets. This is because most hotspots in this dataset are clustered with overlapping coverage.

Observation 2: WiScan reduces 50-62% energy cost of seeking Wi-Fi connectivity compared to existing scans (Figure 9). We have four key findings when comparing WiScan to existing methods. *First*, cutting the scan tax contributes the most on energy saving. While the main processor consumes more than half of the energy in Existing and WiFisense, it consumes less than 1% of energy in WiScan. This indicates

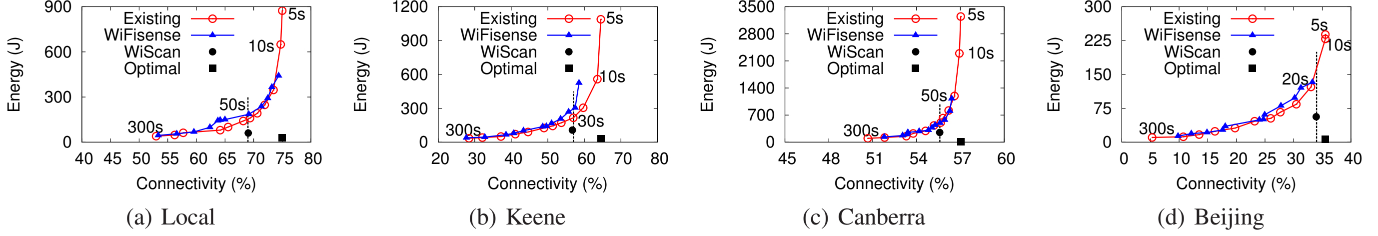


Figure 9. Overall performance of WiScan, WiFisense with different activity sensing frequency (f), and existing scan with different scan interval (T_s). WiScan achieves 90%+ of the optimal connectivity and reduces 50%-62% of the energy cost for seeking Wi-Fi connectivity compared to WiFisense and Existing with similar connectivity.

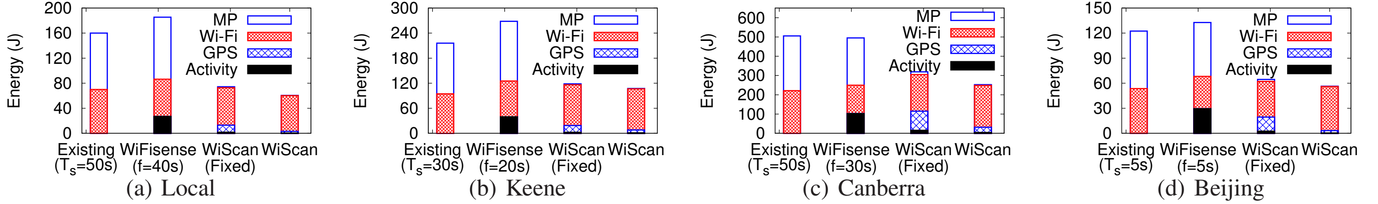


Figure 10. Energy breakdown of Existing, WiFisense, WiScan (Fixed) that uses fixed scan parameters (i.e., scan interval, timeout), and WiScan. We configure the first three so that they achieve nearly the same connectivity as WiScan. WiScan's significant energy saving is from cutting the scan tax (energy consumed by main processor), and adapting scan parameters to reduce unnecessary scans and SSID list computations.

that the cost of computing SSID list and scan parameters in WiScan is negligible. *Second*, adapting scan parameters in WiScan moderately reduces (10-30%) the energy cost. The adaptation achieves higher gain in the Canberra and Beijing dataset, where the available connectivity is lower than other datasets (Figure 9(c)(d)) and users are mostly stationary (88% and 63% of the total duration respectively, Table 3). Hence adapting scan parameters avoids more unnecessary scans and saves more energy. *Third*, the sensing overhead of WiScan is negligible, with GPS as the dominating sensor (87.5% of the sensing overhead). The sensing overhead, however, is compensated by the energy saving of cutting the scan tax. *Fourth*, WiFisense effectively reduces the number of scans, yet it overlooks the energy cost of the main processor and periodical activity sensing. This cost outweighs the energy saving in the Wi-Fi radio, leading to a higher total energy.

Efficacy of SSID Learning Schemes

Next we evaluate different SSID learning schemes, design choices within the Non-uniform scheme, and the impact of SSID list size. We consider ideal SSID list as a reference, the ideal case where we can perfectly predict future SSIDs, or when the Wi-Fi radio's memory can store all public SSIDs.

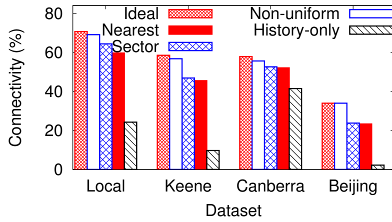
Figure 11(a) compares the connectivity achieved by different SSID schemes to that of the ideal SSID list. We also include the algorithm in Android 4.4.2 factory image, which uses the most recently connected SSIDs, referred to as *History-only*. Windows 8 uses the same SSID selection policy. We make three key observations. *First*, using purely historical SSIDs leads to significant loss of Wi-Fi connectivity, especially for Keene and Beijing datasets where users do not have repeated routes, where it achieves less than one-tenth of the connectivity of the ideal SSID list. Both cases reflect the user traveling case where historical SSIDs are not relevant. This emphasizes that to support diverse user mobility patterns, we need new design of SSID selection policy. *Second*, Non-uniform consistently reaches 96%+ of the connectivity of the ideal SSID list. This verifies the effectiveness of Non-uniform, which incorporates user directionality to better predict future SSIDs. *Third*, compared to the straw-man solutions, Non-uniform achieves larger gain in Keene and Beijing datasets, where hotspots are denser and unevenly distributed and user directionality a critical factor for SSID selection.

We further dive into specific design choices of 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 11(a) shows that 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, while historical SSIDs are only helpful for users with regular mobility patterns (e.g., the Local and Canberra datasets).

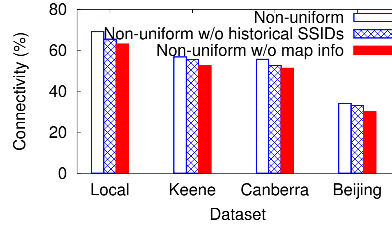
We also examine the impact of SSID list size. We vary the list size from 4 to 60 and calculate the achieved connectivity using Non-uniform. The key observation is that for all datasets, the connectivity quickly converges to that of the ideal list once we offload 16 SSIDs, which is also the maximal number of SSIDs that existing Wi-Fi radio can store. This indicates the efficacy of Non-uniform under the memory constraint of the Wi-Fi radio. We omit the result in the interest of space.

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 12 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



(a) SSID learning schemes



(b) Non-uniform

Figure 11. Impact of SSID schemes, decision choices within Non-uniform, and SSID list size on network connectivity. Non-uniform consistently achieves 96%+ of the connectivity of an ideal SSID list, and outperforms other schemes.

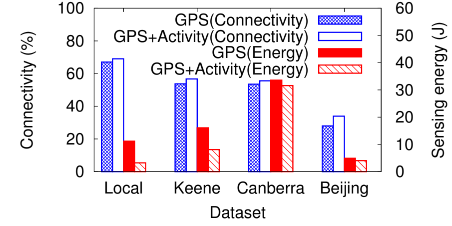


Figure 12. Impact of sensing on connectivity and energy cost. Adding sensors reduces overall energy by cutting the number of updates on SSID list and scan parameters.

calculating scan interval. Adding activity sensors marginally improves the achieved connectivity and surprisingly leads to lower energy cost. The reason is two-fold. First, activity sensors (e.g., accelerometer) consumes low power, much lower than that of GPS. Second and more importantly, adding activity sensors fine-tunes the calibration of scan parameters. As a result, it reduces the times when the number of offloaded scans hits the timeout value. This leads to fewer updates of the SSID list, scan parameters, and GPS data, and reduces the energy cost. For Local and Keene datasets, adding activity sensors achieves much higher energy saving (70%+ and 50% respectively). This is because users in these two datasets are non-stationary for a higher percentage of time (Table 3) and the calibration of speed estimation leads to higher energy saving. Overall adding sensors is beneficial: it improves connectivity while reducing the total sensing cost.

RELATED WORK

Prior work has offloaded partial upper-layer protocol to hardware either to boost performance [31, 39] or to save energy [38]. The offloaded protocols range from TCP/IP stack [31, 39] to ARP and ICMP [38] protocols. These designs introduce a secondary processor to perform the offloaded tasks. WiScan differs in that it uses the existing microprocessor on the Wi-Fi radio without introducing a secondary processor. We explore offloading a Wi-Fi protocol task (scan) to the radio microprocessor, aiming to seek maximal Wi-Fi connectivity with low power.

Active research has examined Wi-Fi energy efficiency in the connected state [41, 42, 47, 57, 62]. Our work complements them by examining the disconnected state where considerable energy saving is possible. To save Wi-Fi scan energy, existing work [33, 43, 50, 60, 64] reduces the number of scans, using either cellular/Bluetooth information [33, 60], or optimized scan intervals [43], or connection history [50], or Wi-Fi beacon patterns [64]. Yet they all overlook the main processor's energy, which is the root cause of the scan energy inefficiency.

Existing work [33, 40, 43, 51] on predicting Wi-Fi connectivity provides valuable insights for our algorithmic designs. Yet they overlook the energy associated with the main processor, work only for users with a regular mobility pattern, and require frequent periodical location sensing to achieve accurate prediction. Our design executes prediction algorithms only when the main processor is active. It computes the SSID list and scan-related parameters on the fly while minimizing the sensing overhead. Prior work on mobility prediction [32, 44, 53, 63] either require additional devices or entail heavy computational overhead. WiScan uses a lightweight mobility prediction scheme to minimize the energy consumption.

CONCLUSION AND FUTURE WORK

We studied the Wi-Fi scan tax problem, the energy inefficiency of existing Wi-Fi scan implementation. We presented WiScan to cut the scan tax by fully exploiting the benefits of scan offloading. Our results validated that WiScan enables ultra-low power hunting for Wi-Fi connectivity, critical for future context-aware apps requiring always-on connectivity.

We also recognize the limitations and possible extensions of our study. *First,* our current design assumes perfect knowledge of hotspot locations. We will study the impact of inaccurate or incomplete hotspot database on WiScan performance and possible design enhancement. *Second*, We plan to extend WiScan to other mobile platforms (e.g., Windows). Implementing WiScan requires the firmware support that recent Wi-Fi chipsets [14, 19] already provide, and modifications at the OS and driver level. We also plan to explore WiScan on other smart devices, especially wearable devices with tight energy budget. *Third*, we plan to examine offloading BSSIDs to Wi-Fi radio for better security. BSSIDs are unique identifiers of APs. Thus it is crucial to design effective schemes that select a small set of BSSIDs to offload. The principle of our SSID learning scheme still applies. We plan to further examine the selection metric. *Fourth*, our current prototype uses GPS for localization. It raises concerns on the energy cost and its efficacy in indoor scenarios. We are interested in integrating low-power alternatives [45, 52, 54, 56, 61] for indoor localization. *Fifth*, when a device roams across APs, the link setup and association can take more than 6 s. We plan to examine features in the 802.11 ai [5] protocol, which speeds up the link setup for Wi-Fi roaming and reduces it to less than 1 s. *Finally*, the number of available networks will increase with the deployment of carrier Wi-Fi networks [3, 4] and long-range white space networks [29, 30]. We plan to extend WiScan's methodology to other types of radio networks, seeking ubiquitous network connectivity with low energy.

Acknowledgment

We sincerely thank reviewers for the insightful comments. We also thank DartNets lab members Rui Wang and Fanglin Chen for their support on this study. This work is supported in part by the Google Faculty Research Award Program.

REFERENCES

1. http://crawdad.org/uiuc/uim/.
2. http://crawdad.org/cmu/hotspot/.
3. http://www.cablelabs.com/carrier-grade-wi-fi-keeps-pace-with-wi-fi-network-growth-how-cablelabs-is-contributing/.
4. http://www.fiercewireless.com/tech/story/confirmed-hotspot-20-comcasts-roadmap-its-xfinity-wi-fi-network/2014-04-11.
5. 802.11ai. http://www.ieee802.org/11/Reports/tgai_update.htm.
6. Android Kit-Kat 4.4. http://www.android.com/versions/kit-kat-4-4/.
7. CRAWDAD. http://crawdad.cs.dartmouth.edu/.
8. FreedomPop. http://www.freedompop.com/.
9. Google Activity Inference. http://developer.android.com/reference/com/google/android/gms/location/ActivityRecognitionClient.html.
10. Google Geocoding API. https://developers.google.com/maps/documentation/geocoding/.
11. IFIXIT.com. http://www.ifixit.com/Teardown/.
12. Jiwire. http://www.jiwire.com/.
13. LinkNYC. http://www.link.nyc/.
14. Marvell Avastar 88W8787. http://www.marvell.com/wireless/avastar/88W8787/.
15. Microsoft Wi-Fi. https://www.microsoftwifi.com/.
16. Monsoon Solutions Inc. http://www.msoon.com/.
17. NLO in Windows. http://msdn.microsoft.com/en-us/library/windows/hardware/hh440295(v=VS.85).aspx.
18. Project Fi. https://fi.google.com/about/.
19. Qualcomm Atheros AR6003. http://www.qca.qualcomm.com/technology/technology.php?nav1=47&product=67.
20. Republic Wireless. https://republicwireless.com/.
21. San Francisco WiFi. http://www6.sfgov.org/index.aspx?page=246.
22. wefi. http://www.wefi.com/.
23. Wireless Minneapolis. http://www.ci.minneapolis.mn.us/wireless/.
24. The future of hotspots: Making Wi-Fi as secure and easy to use as cellular. White paper, Cisco, 2011.
25. The Era of Ubiquitous Listening Dawns. MIT Tech Review, 2013.
26. What Apples M7 Motion-Sensing Chip Could Do. MIT Tech Review, 2013.
27. Boston spreads free Wi-Fi hotspots. The Boston Globe, 2014.
28. When wireless worlds collide. The Economist, 2014.
29. Green light for 'TV white space' wireless technology. Ofcom for Consumers, 2015.
30. Microsoft Starts Slashing African Internet Prices with White-Space Networks. MIT Tech Review, 2015.
31. Agarwal, Y., et al. Somniloquy: augmenting network interfaces to reduce PC energy usage. In *Proc. of NSDI* (2009).
32. Alvarez-Lozano, J., García-Macías, J. A., and Chávez, E. Learning and user adaptation in location forecasting. In *Proc. of UbiComp* (2013).
33. Ananthanarayanan, G., and Stoica, I. Blue-Fi: Enhancing Wi-Fi performance using Bluetooth signals. In *Proc. of MobiSys* (2009).
34. Anton, B., Bullock, B., and Short, J. Best current practices for wireless internet service provider (WISP) roaming, version 1.0. *Wi-Fi Alliance* (2003).
35. Bahl, P., and Padmanabhan, V. RADAR: an in-building RF-based user location and tracking system. In *Proc. of INFOCOM* (2000).
36. Bertram, J. E. A., et al. Multiple walking speedfrequency relations are predicted by constrained optimization. *Journal of Theoretical Biology* (2001), 445–453.
37. Chen, Y., et al. FM-based indoor localization. In *Proc. of MobiSys* (2012).
38. Christensen, K. J., et al. The next frontier for communications networks: power management. *Computer Communications 27*, 18 (2004), 1758–1770.
39. Currid, A. TCP offload to the rescue. *Queue 2* (May 2004), 58–65.
40. Deshpande, P., et al. Predictive methods for improved vehicular WiFi access. In *Proc. of MobiSys* (2009).
41. 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).
42. Garcia-Saavedra, A., et al. Energy consumption anatomy of 802.11 devices and its implication on modeling and design. In *Proc. of CoNEXT* (2012).
43. Kim, K.-H., et al. Improving energy efficiency of Wi-Fi sensing on smartphones. In *Proc. of INFOCOM* (2011).
44. Koehler, C., Banovic, N., Oakley, I., Mankoff, J., and Dey, A. K. Indoor-alps: an adaptive indoor location prediction system. In *Proc. of UbiComp* (2014).
45. Liu, H., Darabi, H., Banerjee, P., and Liu, J. Survey of wireless indoor positioning techniques and systems. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on 37*, 6 (2007), 1067–1080.

46. Liu, J., et al. Energy efficient GPS sensing with cloud offloading. In *Proc. of SenSys* (2012).

47. Liu, J., and Zhong, L. Micro power management of active 802.11 interfaces. In *Proc. of MobiSys* (2008).

48. Long, L. L., and Srinivasan, M. Walking, running, and resting under time, distance, and average speed constraints: optimality of walk-run-rest mixtures. *Journal of The Royal Society Interface 10*, 81 (2013).

49. Megiddo, N., and Modha, D. ARC: A self-tuning, low overhead replacement cache. In *Proc. of FAST* (2003).

50. Navda, V., et al. MobiSteer: Using steerable beam directional antenna for vehicular network access. In *Proc. of MobiSys* (2007).

51. Nicholson, A. J., and Noble, B. D. BreadCrumbs: forecasting mobile connectivity. In *Proc. of MobiCom* (2008).

52. Otsason, V., Varshavsky, A., LaMarca, A., and De Lara, E. Accurate GSM indoor localization. In *Proc. of UbiComp*. 2005.

53. Patterson, D. J., Liao, L., Fox, D., and Kautz, H. Inferring high-level behavior from low-level sensors. In *Proc. of UbiComp* (2003).

54. Popleteev, A. Device-free indoor localization using ambient radio signals. In *Proc. of UbiComp* (2013).

55. Ra, M.-R., et al. Improving energy efficiency of personal sensing applications with heterogeneous multi-processors. In *Proc. of UbiComp* (2012).

56. Rabaey, J. M., Ammer, M. J., da Silva Jr, J. L., Patel, D., and Roundy, S. Picoradio supports ad hoc ultra-low power wireless networking. *Computer 33*, 7 (2000), 42–48.

57. Rozner, E., et al. NAPman: network-assisted power management for WiFi devices. In *Proc. of MobiSys* (2010).

58. Scellato, S., et al. NextPlace: A spatio-temporal prediction framework for pervasive systems. In *In Proc. of Pervasive* (2011).

59. von Nagy, A. Wi-Fi alliance rebrands Hotspot 2.0 as Wi-Fi certified passpoint. `http://www.revolutionwifi.net/2012/05/wi-fi-alliance-rebrands-hotspot-20-as.html`, 2012.

60. Wu, H., et al. Footprint: Cellular assisted Wi-Fi AP discovery on mobile phones for energy saving. In *Proc. of WINTECH* (2009).

61. Xie, H., Gu, T., Tao, X., Ye, H., and Lv, J. Maloc: a practical magnetic fingerprinting approach to indoor localization using smartphones. In *Proc. of UbiComp* (2014).

62. Zhang, X., and Shin, K. G. E-MiLi: energy-minimizing idle listening in wireless networks. In *Proc. of MobiCom* (2011).

63. Zheng, Y., Li, Q., Chen, Y., Xie, X., and Ma, W.-Y. Understanding mobility based on GPS data. In *Proc. of UbiComp* (2008).

64. Zhou, R., et al. ZiFi: Wireless LAN discovery via ZigBee interference signatures. In *Proc. of MobiCom* (2010).