A Feasibility Study of Distributed Spectrum Sensing using Mobile Devices

A Thesis presented

by

Udit Kumar Gupta

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

May 2015

Stony Brook University

The Graduate School

Udit Kumar Gupta

We, the thesis committe for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis

Samir Das
Professor, Computer Science

Himanshu Gupta
Associate Professor, Computer Science

Aruna Balasubramanian
Assistant Professor, Computer Science

This thesis is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract

A Feasibility Study of Distributed Spectrum Sensing using Mobile Devices

by

Udit Kumar Gupta

Master of Science

in

Computer Science

Stony Brook University

2015

Given the exponential increase in mobile data traffic, there is a growing fear of an impending spectrum crunch. Shared use of the so-called 'Whitespace' spectrum offers a solution. Whitespace spectra are those that are already licensed for specific use but is understood to be ill-utilized over space and time. Examples include spectra used by terrestrial TV broadcast or various radars. Shared use of such Whitespace spectra requires that incumbent licensed devices (primary) would have priority. One straightforward mechanism to detect the existence of primary transmissions is spectrum sensing. We are envisioning a spectrum sensing model where mobile devices have built-in spectrum sensing capabilities and upload such data on a cloud-based server that in turn builds a spatial map of spectrum occupancy. This enables spatial granular data collection via crowd-sourcing, for example.

However, such mobile device-based spectrum sensing is challenging as the mobile devices are generally resource constrained. On the other hand, spectrum sensing is energy and computation intensive. In this thesis, we perform a measurement study to understand the general feasibility of the mobile sensing approach. For the experiments, we use several low-power software radio platforms that are powered by USB so that they can be interfaced to a mobile phone/tablet class device with an appropriate USB support. We develop appropriate software/hardware testbeds to carry out latency and energy measurements for mobile-based spectrum sensing on such platforms for a suite of well-known sensing algorithms operating in the TV white space. We describe the setup, discuss insights gained from these measurements and different possible optimizations such as the use of pipelining or use of GPU. Finally, we demonstrate the end-to-end operation by showcasing a system comprising of i) a number of distributed mobile spectrum sensors and ii) an indoor small cell comprising of an access point and client device (secondaries) operating in TV band iii) and a cloud-based spectrum server that builds spectrum map based on collected sensing data and instruct/allow secondaries to operate in multiple non-interfering Whitespace channels based on availability at the location.

Contents

List of Figures

List of Tables

Abbreviations

CAGR	**C**ompound **A**nnual **G**rowth **R**ate
MSS	**M**obile **S**pectrum **S**ensor
DSA	**D**ynamic **S**pectrum **A**ccess
AP	**A**ccess **P**oint
ACF	**A**utocorrelation
SDR	**S**oftware **D**efined **R**adio
GND	**G**round
NDK	**N**ative **D**evelopment **K**it
FCC	**F**ederal **C**ommunications **C**ommission
IDW	**I**nverse **D**istance **W**eighting
REM	**R**adio **E**nvironment **M**ap
FFT	**F**ast **F**ourier **T**ransform

Acknowledgements

First and foremost, I thank Prof. Samir Das for his continuous guidance and feedback over the course of my research. It was first week of Fall 2013 when I first walked into his office, inquiring about a independent research opportunity. I have learned so much since then, making this experience really valuable for me.

I also thank Prof. Sumit Roy from University of Washington for all his support during this project. The subject of this thesis was challenging not only because of the building a functional system architecture for Whitespaces but also because of the analytical methods involved for processing spectrum measurement data. He helped us understand the challenges of spectrum sensing which allowed me to rethink and implement the benchmarking testbed for sensing devices and sensing algorithms.

Many thanks to Stony Brook University and to the Department of Computer Science for the financial aid they provided and their ongoing support during last year of my graduation. Finally I thank CEWIT laboratory for using their hardware labs time to time. This work was also funded by NSF as a collaborative research with University of Washington.

Likewise, I am grateful for the supervision I received from my seniors in this group, who were (in chronological order) Ayon Chakroborty and Jihoon Ryoo, along with the rest of my lab-mates in the WINGS (Wireless Networks and Systems Laboratory) Group. Last but definitely not least, I am forever grateful to my friends and family for their support.

Chapter 1

Introduction

With the continuous growth in the number of smartphone users worldwide, there is an exponential growth in global mobile data traffic as well, and it is expected to grow even more exponentially in coming years. As per [1], global mobile data traffic will increase at a compound annual growth rate (CAGR) of 57 percent from 2014 to 2019, reaching 24.3 exabytes per month by 2019. Also, the number of mobile-connected devices are soon expected to exceed the number of people on earth [1]. It can potentially lead to the problem of spectrum crunch (the lack of sufficient frequencies to support the data needs of mobile devices).

Dynamic Spectrum Access (DSA) [2] [3] [4] [5] offers a solution to this problem by describing set of technologies where multiple alternate and underutilized radio spectrum bands can be opportunistically used to transmit data. Next generation self-configuring networks are being envisioned where a network can operate in multiple wireless technologies in different frequency bands. Users can exploit these bands as per the data need in an opportunistic manner. The original Licensed users of these spectrum bands are referred to as primary users [6] while secondaries are potential users of the available free spectrum (spectrum holes or Whitespaces) where no transmission is taking place by primary users. As soon as primary users begin transmitting on same channel/frequency, secondary must leave the system so as to avoid the interference. It opens up the challenge of monitoring these Whitespaces by secondary devices in an efficient manner. TV Whitespace spectrum is the first widely accepted opportunity that can be used for dynamic spectrum sharing and many systems/techniques are already in place to monitor the spectrum holes for secondary use. In next section, we describe TV Whitespace in detail.

1.1 TV Whitespace

TV Whitespace refers to the unused or underutilized frequencies in VHF and UHF television broadcast spectrum/band that can be used for the dynamic spectrum access purpose. Not all of channels operating at these frequencies are used at same time/location. This makes TV Whitespace an excellent choice for dynamic spectrum access.

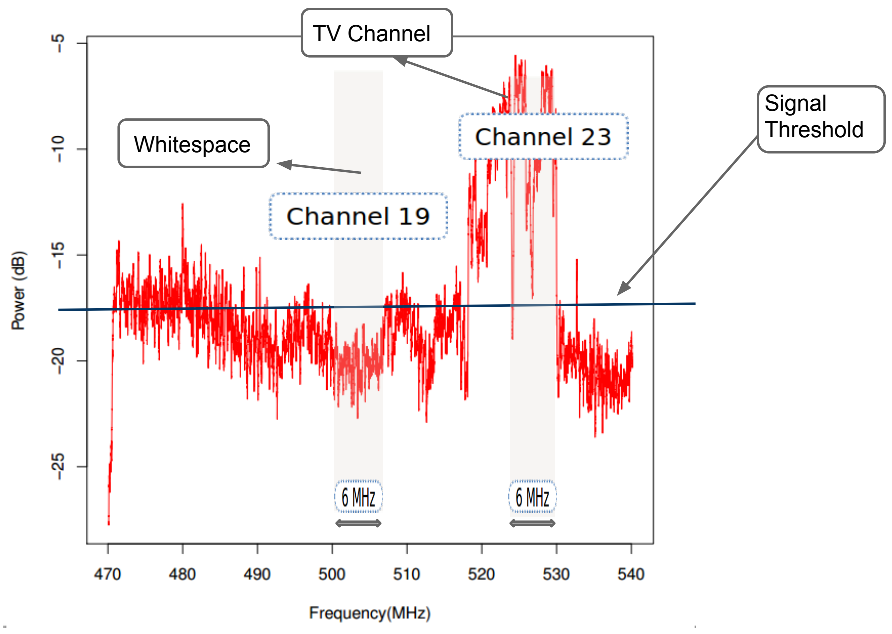


FIGURE 1.1: Anatomy of an TV signal

In the United States, TV signal follow ATSC standards [7] and are divided into two bands - VHF and UHF. Most of this TV Whitespace in the US lies in UHF band. Recent FCC ruling [8] has also allowed access to these television frequencies in a shared opportunistic manner keeping primary[1] to be on priority. The UHF band available for shared use contains channel 14 to channel 51, starting from 470 MHz as lower frequency edge. Each channel width is 6 MHz and a channel presence, in general, can be detected by an ATSC Pilot at 300 KHz away from the lower edge of the channel. Figure 1.1 show a real scan results using RTL-SDR TV dongle over the UHF TV band in the Stony Brook city, New York. The presence of a TV signal (or primary) can be detected by higher power value over the threshold. An high power value means primary is transmitting over this channel, and any secondary device can not use this channel. We need to find those holes or Whitespaces where the TV signal power is so low that it can not interfere with the primary station. In Figure 1.1, channel 19 is available for use as Whitespace while channel 23 is being used as TV channel and can not be used for secondaries.

[1]Incumbent spectrum users like TV broadcasters, public safety, microphone users, etc are some examples of primary TV band devices.

Channel Number	Frequency Range(MHz)
5	76-82
6	82-88
26	542-548
51	692-698

Table 1.1: Whitespace Channel Availability for TV Spectrum in Stony Brook, as given by Spectrum Bridge

We need mechanisms to exploit these TV Whitespaces in an efficient manner. Availability of these Whitespaces has been found to be very precious in space and time but discovering them is very hard without actual spectrum sensing. As seen in Table 1.1 and 1.2, Spectrum Bridge(in time) [9] lists 4 TV channels as Whitespace (out of 51) while Google Spectrum Database(in space)[10] marks the same information on the map and lists only two channels (channel 5 and 6) for the city of Stony Brook in New York State. The work in [11] also highlights that the Whitespace presence has an inverse relation to the population density where the potential spectrum crunch probability is highest. This makes the problem even more challenging so we can not afford to lose the spectrum awareness information due to some faulty techniques. We need to come up with effective spectrum measurement procedures that would allow us to monitor the spectrum in real-time, are less error prone and also enable us to detect the unauthorized spectrum usage as well.

1.2 Measuring Whitespaces in Traditional Approach

The recommended solution by Federal Communications Commission(FCC) to measure the TV band spectrum occupancy is to use online spectrum databases that provides a list of Whitespace channels available for secondary use at a given location. Figure 1.1 and 1.2 show available TV Whitespace channels, as obtained from two such popular databases - Spectrum Bridge [9] and Google Spectrum Database [10]. However, these propagation modeling based databases provide poor estimates of Whitespace and are prone to many errors, especially in urban areas[12] [13] [14] [15]. These databases estimates the protection regions [2] for each TV channel based on certain TV transmitter attributes and some well-known statistical models [3], although some attributes are known to significantly influence the database accuracy (like use of average terrain instead on

[2]Secondary devices are allowed to transmit only outside this area

[3]F-curves [16] in case of SpectrumBridge, as recommended by FCC

fine -grained terrain for SpectrumBridge) [17]. Further, many of the measurement works in recent past have also pointed out various limitations of using propagation models for predicting Whitespace channel availability [18] [19] [12] [13] [14] [15]. We need to perform actual spectrum sensing and populate/augment these databases with real measurements data [17] for better realization about Whitespace channel availability.

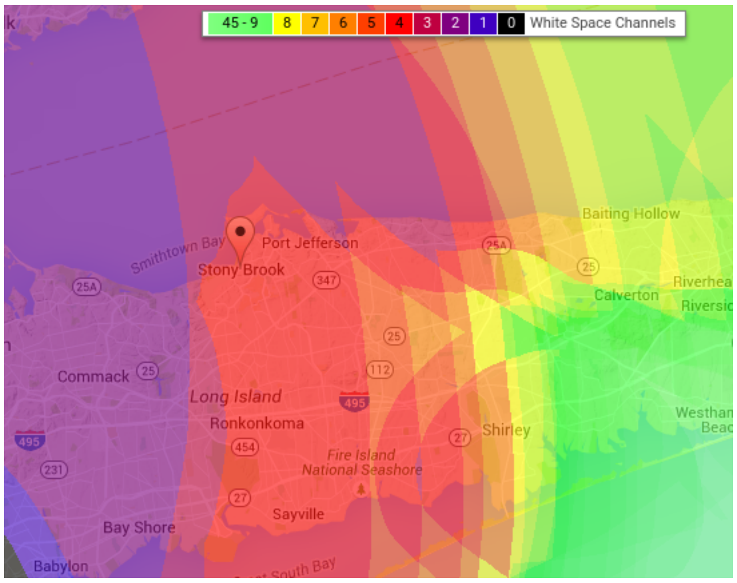


FIGURE 1.2: TV Whitespace Availability for Stony Brook Area, as given by Google Spectrum Database

Existing solutions for real-time spectrum monitoring majorly rely on static spectrum sensors deployment. Many well-known system designs are already in place which put these static sensors at predefined locations and network them together to bring data to a central repository for generating spectrum occupancy map [20] or radio environment map(REM). However, the the bulky and costly [4] nature of these spectrum sensors makes them infeasible to use for real-time spectrum monitoring applications where we need high granular measurements, as in case of detecting TV Whitespace availability [22] [23] [24]. Recent works [15] have also deployed similar sensors in public vehicles to collect real-time data so as to validate the location of primary and secondary devices and determine the channel quality. Also, there are systems in place that talk about deploying these sensors at access points locations for exploiting TV Whitespaces [25]. All these solutions do not provide enough granularity for evaluating the spatio-temporal usage patterns of TV spectrum and detecting unauthorized signal presence in the context of TV Whitespace. We need more granular measurements in an efficient manner for better spectrum awareness. Using commodity mobile devices for spectrum sensing purpose

[4]For e.g. Rohde and Schwarz, FSU50-B25-K40 spectrum analyzer costs about $ 75000 [21], more than two orders of magnitude compared to USRP B210, the most expensive device in our study and almost three orders of magnitude higher than an RTL-SDR dongle, the least expensive device in this work

(called mobile spectrum sensing) provides a viable alternative leveraging the ubiquitous nature of these devices. We discuss mobile spectrum sensing in detail in next section.

1.3 Our Approach: Mobile Spectrum Sensing

We discussed in last section that there is a need for fine-grained real-time data collection/measurements for better spectrum management that is not feasible with the traditional measures of Whitespace occupancy/detection techniques. Mobile devices are ubiquitous in nature and have the advantage of going everywhere. Mobile spectrum sensing thus envision a future scenario where these mobile devices have spectrum sensing ability and can report the real-time measurement data to a central agency (cloud-based server) that can build a spectrum occupancy map based on the collected data. This can enable highly granular data collection with crowdsourcing like techniques and can be very useful, particularly in situations where the need for spectrum sensing is high. Availability of these TV Whitespaces infrequently changes over space, so we do not need to distribute the task of spectrum sensing to all sensing devices at a location, rather only few devices in a single geographical region may serve the purpose. Spectrum sensing can also be used in conjunction with databases with only modest amount of measurements [17]. TV Whitespace availability is also infrequent in time. So, we do not need spectrum information every time. Rather, mobile sensors can be asked to upload information after some regular interval of time. Further, [26] also discusses that using mathematical prediction algorithms techniques like interpolation [5], mobile sensors can perform the sensing task with similar accuracy in an infrequent spatial manner.

Looking at all this information, mobile spectrum sensing seems like a good alternative to explore for measuring Whitespace availability. These mobile devices currently lack appropriate interfaces to carry out the spectrum sensing task in the context of TV Whitespace, although some prototypes are already in place for building such a system. [27] [28].

Recent advances in low-power [6] low-cost [7] software defined radios (SDRs) have enabled us to make a case for spectrum sensing task on mobile platforms with the help of small form factor sensing devices. These low-cost small form factor sensing devices can be

[5]Interpolation is a numerical method for getting values at positions in between a known discrete set of points. In the context of mobile spectrum sensing, interpolation can be used to predict the power values at unknown points using only a set of known data points within an area

[6]The low-power here refers to the power consumed by radio front-end of SDRs in an idle state when it is doing no useful work. We also verified that a typical RTL-SDR dongle device consumes two orders of magnitude less power compared to ThinkRF - a well-known wideband spectrum analyzer [29], and about three orders of magnitude less power compared to typical Agilent ESA E-series spectrum analyzer [30], a popular industry standard for spectrum sensing.

[7]Compared to traditional static sensors discussed in last section

integrated with commodity mobile devices to realize a mobile spectrum sensor(MSS) ([26]. The work in [31] shows the statistics that putting lots of noisy low-cost sensors can achieve similar accuracy compared to the high-end, costly sensors at few locations.

However, these mobile devices are resource constrained units and have limited computation capabilities. Sensing algorithms (or spectrum occupancy detection algorithms, discussed in Chapter 3) may consume more time when running on these mobile platforms which eventually makes the decision of spectrum occupancy go slower. Spectrum sensing could also consume significant processing power as the radio front end could consume significant energy so the feasibility of the whole idea could be uncertain as well. So, these mobile spectrum sensors although seems to make a good case as a replacement for costly, bulky and infrequent spectrum analyzers but we need to ensure that mobile devices don't run out of power soon and report/upload spectrum sensing data to the central cloud-based server in a timely fashion so as to make a case for realistic scenario.

There are several other concerns regarding these low-cost low-power SDRs in general in the context of mobile spectrum sensing. These low-cost SDRs are inherently noisy and vary widely in different performance characteristics. The complete task of spectrum sensing may also become slow in nature due to constrained sensing capabilities of these devices. For example, A sensing device with low sampling rate scans less amount of spectrum at once so it takes higher amount of time to scan a particular frequency range as it needs to sweep the whole spectrum sequentially. So, it is important to understand the various performance aspects of mobile spectrum sensing.

In this work, we do an extensive performance evaluation study for our mobile spectrum sensor prototype utilizing modern small form factor and USB powered software defined radios(SDRs). These off-the-shelf SDRs can be attached to mobile class of devices (Smartphones, low-power development platforms, tiny computers and embedded devices, etc) with appropriate USB support. We perform latency and energy measurements on these mobile platforms for well-known sensing algorithms in TV Whitespace to detect the TV signal with the help of suitable hardware and software testbeds. We also perform similar experiments for USB powered SDRs to evaluate the performance of these low-cost radios. We describe the experimental setup, associated challenges and present our results from these measurements. We further discuss memory level optimizations to save CPU cycles that improves the responsiveness of the system and further discuss possible improvements utilizing mobile GPUs and applicability of similar systems in wide-band sensing. Finally, we demonstrate such a system integrating mobile sensors into a communication network operating in TV white-space band. The thesis exhibits the feasibility of mobile spectrum sensing in real-time practical scenarios in a resource constrained environment utilizing modern mobile platforms and low-cost USB powered software radios.

1.4 Thesis Organization

The rest of the report is organized as follows:

- Chapter 2 discusses related work in the area of spectrum sensing and its performance evaluation on existing platforms in the context of TV Whitespace.
- Chapter 3 discusses the design of a potential mobile spectrum sensor. We discuss how we can form a mobile spectrum sensor by attaching a sensing and compute unit in series. We also discuss different available choices and reasons to use them as a compute or sensing unit in our study.
- Chapter 4 provides details about our testbed setups for various measures of latency and energy related to the sensing and compute units. We explain our experimental setups in detail, layout technical challenges, share evaluation results from our observations and discuss arguments for the existing behavior.
- Chapter 5 discusses possible improvements and optimizations with the help of a pipe-lining approach that can further improve the responsiveness of a compute unit. We also discuss and explain our methodology for exploiting mobile phone GPU's for spectrum sensing purposes that can further optimize the latency of the overall system. We conclude the chapter by listing different factors affecting the performance of a compute unit in general.
- Chapter 6 demonstrates a system comprising of distributed mobile spectrum sensors and a cloud-based spectrum server that build spectrum maps based on collected sensing data and make decisions so as to detect the interference at secondaries location. It further allows Whitespace AP and client(secondaries) to dynamically switch to a non-interfering channel, available in their cell/area in the presence of an interferer that generates the traffic in same frequency/channel in which the Whitespace AP and client were previously communicating with each other).
- Chapter 7 concludes the thesis and discusses the future work.

Chapter 2

Related Work and Research Scope

In academic research, while work has been done on various aspects related to TV Whitespace and spectrum sensing, still there is a lack of work on providing complete end-to-end system for achieving the spectrum measurement in a realistic cost effective manner. Many techniques have been proposed for fast, real-time and energy efficient spectrum sensing [32] [33] [34] [35] [36] but performance realization of these systems has been limited to specialized purpose system only, very few attempts are there for generic performance measurements of these software radios. Some works have talked about embedding sensing functionality into receiver designs but they have failed to provide a complete realistic motivation to invest and build in such an ecosystem to achieve the task [20] [12] [25] [37].

Our work does not talk about any specialized form of sensing in general like narrow-band sensing [38] [39] [40] and wideband sensing [32] [41] [42] [36], rather we seek to evaluate a framework for mobile spectrum sensing which can be applicable among multiple domains of sensing and multiple type of devices. We also do not provide any novelty by attaching low-costs sensing units with mobile devices [26], rather we are going one step forward utilizing this platform along with other SDR devices and evaluating the platform for various performance aspects to look at the feasibility of mobile spectrum sensing support in general for future mobile devices.

In the context of performance evaluation, while there are numerous attempts for the power and energy evaluation of mobile phones [43] and cellular networks [44], but there is still less focus for the comprehensive evaluation study of spectrum sensing on mobile devices. Some works proposes design of new software radio platforms with hardware enhancements. [45] proposes the design of a software radio for the power, size and cost optimization using flash based FPGAs while [46] proposes to develop a similar system using highly-integrated radio front-ends and mixed-signal FPGA processing back-ends, to be operated on a lithium battery. In our work, we do not talk about any hardware redesign for these software radios but seek to evaluate and compare the performance of these so called FPGA and Non-FPGA devices in general. The work in [47] although claims to motivate the feasibility of low-cost and low-power sensing for versatile mobile devices, it is restricted to the evaluation of sensing performance using analog SDR front-end and sensing functionality for DVB-T only. The work in [27] integrates a spectrum sensor for the detection of primaries with old Nokia N900 phones, but concludes with the need for the more practical testing with smart phone size devices to realize an mobile spectrum sensor which is the central component of our work. A recent system [48] also provides a prototype system for translating one's mobile phone into a spectrum analyzer using WiFi to collect spectrum measurements, and leveraging the frequency translator for extending the sensing range but still the question about the feasibility of mobile spectrum sensing in real time scenario remains the same.

A Recent work [49] provides evaluation study of spectrum sensing on commodity mobile devices. Although both works share the same goal at high level, our work focus on the latency and energy measurements of mobile and sensing devices along with possible optimizations while the work in [49] is mainly focused on signal detection accuracy. Also, their system includes an RTL-SDR dongle along with a smartphone and mobile device and performance have been compared with USRP, we seek to compare the performance of multiple heterogeneous mobile and sensing devices in our work in the context of TV Whitespace.

Work in [50] discusses design of a low-cost sensor platform for wideband sensing and talk about utilizing both CPU and GPU of a laptop like device for the purpose of spectrum sensing. We have used the low-power mobile platforms for our work (including GPU) rather than the laptop. Also, We have done an individual component analysis of CPUs and GPUs for latency and energy measurements while the previous work has shared only the final overall performance results.

The last part of our work - An actual system capable of running in TV Whitespace can be assumed to be similar to the work in [25] which propose a dual technology architecture for wide-area wireless users. It extends the functionality of indoor APs to communicate over two bands — the cellular operator's own band using traditional approach, and the TV Whitespace spectrum for opportunistic use. Although putting complete system in place really motivates the same idea but the paper does not address any performance issue about the mobile spectrum sensors that are the major concern of these mobile sensors and our primary goal is to understand the behavior of such system in general.

These systems can further utilize a crowd-sourcing like scenario for getting spectrum sensing information at a high granularity and allocating task to these sensors in an optimal manner. [51] [52]

In a nutshell, The focus of this work is to perform a feasibility analysis to provide complete end-to-end system design in the UHF TV band using commodity mobile devices. Specifically, this work seeks to produce a sensor prototype that is capable of meeting the realistic scenarios, and that does so with a low-power overhead appropriate for mobile applications. This design is targeted solely as a guideline for next generation receiver front-end design on mobile phones for sensing as well as computation (capable of decoding data) functionality but a fully-functional receiver design is outside the scope of this work.

Chapter 3

Mobile Spectrum Sensor

In this chapter, we propose a logical abstraction for our mobile spectrum sensor. We envision about a future mobile device with spectrum sensing capabilities. We define the behavior of such a device by logically partitioning the functionality (based on individual behavior) into three units: a) sensing unit, b) compute unit and c) a communication interface.

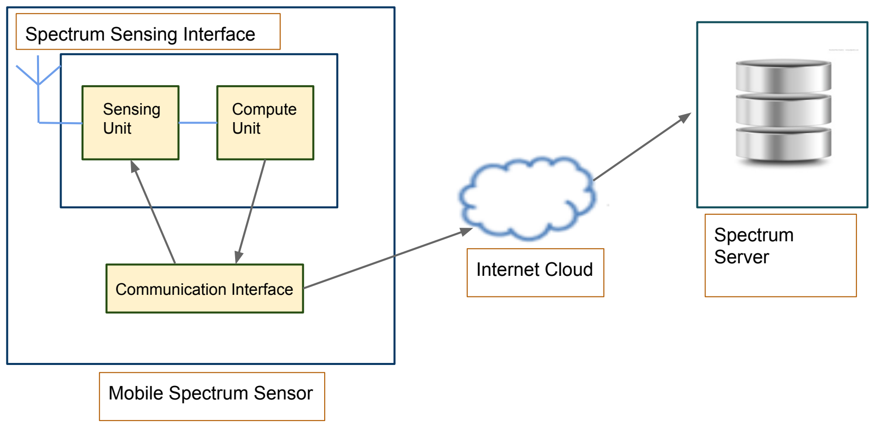


Figure 3.1: Mobile Spectrum Sensor Block Diagram

The sensing unit consists of a radio unit that can scan the desired frequency range (spectrum) and get the digitized IQ samples. These IQ samples are passed to the associated compute unit for further processing. The communication interface can talk to remote cloud-based spectrum server for instruction regarding what spectrum to sense, what parameters to tune to, etc. This interface can also report back the results to the spectrum server. The Compute unit handles the actual processing, to make the most accurate decision than current existing solutions about the spectrum situation at a particular location and time.

3.1 Mobile Spectrum Sensor Architecture

In this section, we discuss in detail about the architecture of our mobile spectrum sensor prototype. We describe our realization of individual units present in this system and then discuss in detail various options considered for sensing and compute unit.

Figure 3.1 shows the block diagram and logical working of a mobile spectrum sensor. To understand it at a very high level, Sensing unit is capable of sensing TV band frequencies (The mobile spectrum sensing is a more general idea than TV band. We are doing TV band as these are the only sensors available to us on USB). It passes IQ samples to the compute unit for further processing. Compute unit process the received IQ samples and makes a decision about Whitespace availability (in TV band). Communication Interface pass instructions obtained (from spectrum server) to the sensing unit, get results from compute unit and pass them back to the spectrum server.

3.1.1 Sensing Unit

We have the vision to have an on-chip sensor embedded in the compute device for spectrum sensing in the future. Although there is no reason for not being able to put together a system with the required chipset having spectrum sensing support on a small form factor computing platform, there exists no practical implementation for these systems. For spectrum sensing evaluation purposes on mobile platforms, we need to develop a testbed and thus we develop a prototype by attaching off-the-shelf SDRs as sensing units externally to the compute unit for performing the task of spectrum sensing. These low-power USB SDRs are low-cost commodity devices, readily available in the market and allow us to make a low-cost sensing unit for our mobile spectrum sensor prototype. We use the same setup over the course of this thesis for all experiments.

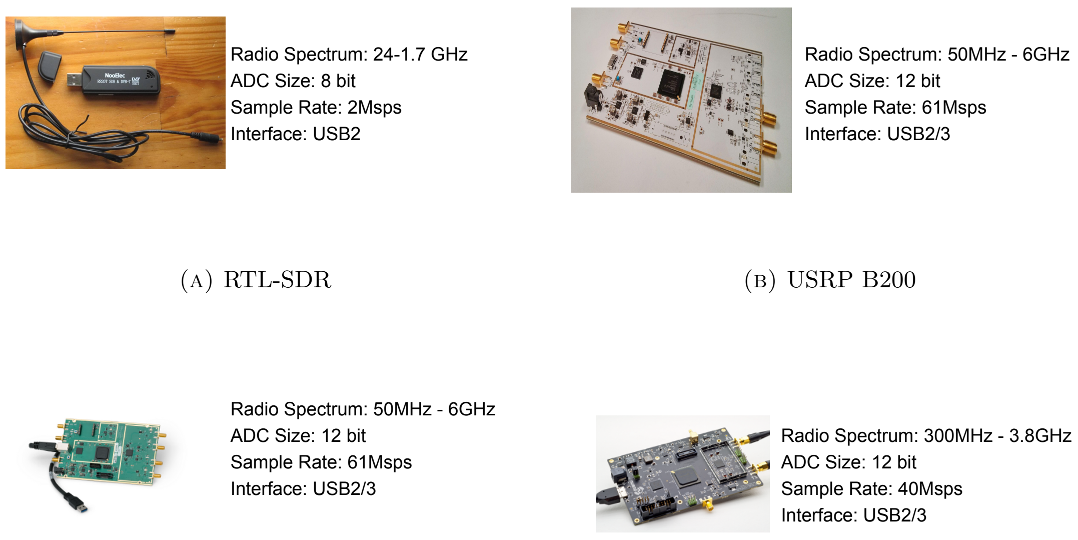
Another main reason to choose these SDRs as sensing units for our work is that these all devices can be powered from USB bus and thus can potentially provide a platform for mobile experiments as almost all of compute platform available today comes with a built-in USB support. We have chosen all such platforms currently available in the market.

For our purposes, sensing unit is nothing but a general software radio capable of scanning a desired frequency range. All these sensing devices are very heterogeneous by nature and have different capabilities, architecture and limitations. Some factors highlighting the heterogeneity among devices are - Resolution, Precision, Sensitivity, Highest Supported Sample Rate, Supported frequency range, Hardware design consideration (FPGA or Non-FPGA), USB bus interface type and latency overhead for individual hardware units, etc. Consider, for e.g. Resolution, it describes the capability of a spectrum sensor to define fine signal intervals. The fine would be the interval, the narrower would be frequency range for a particular band. Similarly, consider the Precision - The USRP B200 device have a 12-bit ADC compared to an RTL dongle device having 8-bit ADC, so it is obviously more precise. Sensitivity of a sensor is the lowest SNR at which a given target probability of error can be encountered. Some devices may have lower sensitivity compared to others. [31] describes these factors in details and their impact on the performance of our system. Figure 3.2 below lists below various SDRs used in this work along with some of their characteristics.

We discuss all of them briefly here for our understanding. In later sections, we conduct experiments based on the intuition developed from this knowledge and tried to make the judgment whether these devices makes up for a case of real mobile spectrum sensor.

- **RTL-SDR:** A Cheap DVB-T Dongle based on the RTL2832U chip, can be used as cheap SDR anywhere. Although low ADC Size may have an impact on the precision, ease in operation and low-cost makes it a suitable spectrum sensor candidate. Also, it can not scan a whole range of frequencies and have low sample rate so suitable only for narrowband sensing purpose, that too limited to a supported radio spectrum range. It operates on USB2 unlike others so may also provide slow data read speed.
- **USRP B200:** An FPGA-based device, highly used and known among the community. Better specifications of this device come with a high cost.
- **USRP B210:** Same as USRP B200 but support high clock precision and known to be more accurate. It also has extra Tx/Rx ports that make it suitable for few applications. It is probably the best sensing device (in terms of accuracy) among all others in this work but this accuracy does not come cheap in hand. (The cost for one unit of this device differs in about 400$ with respect to the USRP B200)
- **BladeRF:** Accuracy of this device is similar to USRP B200, but it comes with a reduced cost. This reduction in cost counts on the low radio spectrum coverage compared to USRP B200.

Although we discussed above that these devices vary in cost, some being highly costly compared to others but these all sensing devices are still much cheaper compared to


Radio Spectrum: 24-1.7 GHz
ADC Size: 8 bit
Sample Rate: 2Msps
Interface: USB2

(A) RTL-SDR

Radio Spectrum: 50MHz - 6GHz
ADC Size: 12 bit
Sample Rate: 61Msps
Interface: USB2/3

(B) USRP B200

Radio Spectrum: 50MHz - 6GHz
ADC Size: 12 bit
Sample Rate: 61Msps
Interface: USB2/3

(C) USRP B210

Radio Spectrum: 300MHz - 3.8GHz
ADC Size: 12 bit
Sample Rate: 40Msps
Interface: USB2/3

(D) BladeRF

FIGURE 3.2: Various SDRs used as Sensing Units

those used by the industry for real scanning purposes and we can completely rely on them for our sensing purpose.

3.1.2 Compute Unit

The second component of our mobile spectrum sensor is the compute unit. Compute unit takes raw IQ samples from the sensing unit and process them to remove the associated noise and calculate the power spectrum density (power present at different frequencies in the band of interest). This unit runs sensing algorithms on the obtained IQ samples to determine the presence or absence of TV signal or Whitespace. Like sensing unit, compute unit can also be heterogeneous, but the factors describing differences among them are somewhat different. Different compute unit are various commodity mobile devices (such as smartphones or tablets), open hardware boards or low-power compute platforms (Raspberry PI, Beaglebone black, Humming board, etc) that could be battery powered and can be used as a proxy for a mobile device. Modern mobile devices have high-end CPUs as well as many of them come equipped with GPUs. These architectural benefits can also be exploited for latency improvements on these low-power devices. We discuss them more in Chapter 4.

In this section, we look at various options we have considered for our compute unit and heterogeneous parameters that describe them. Some factors that describes heterogeneity among compute devices are CPU clock rate, number of CPU cores, presence of graphical cores (GPUs), RAM size, available storage and powering scenario (battery or external power source).

We also look at briefly over the different spectrum sensing algorithms for reporting spectrum occupancy. Later, we look at the performance aspects of these algorithms for various compute units in discussion.

First, we list here different compute units used for our experiments. Figure 3.3 also describe all of them along with specifications of interest.

- **Nexus 5:** One of the high-end smartphone available in market makes a perfect case for a mobile compute unit. The phone has Quad-core Krait 40 processor with 2 GB physical memory and Adreno 330 GPU support. Although Google provided limitations on the use of OpenCL(for GPU) and non-removable battery makes it difficult to use for certain experiments. The phone also have the USB OTG [1](On-The-Go) support unlike its previous version Nexus 4 where the kernel does not support it.
- **Samsung Galaxy S4:** Another good smartphone equipped with a quad-core 1.6GHz CPU along with 2GB RAM and USB OTG support. The phone also have Adreno 320 GPU operating at 400MHz frequency and a removable battery and makes it an excellent choice for experiments that were either not suitable or atleast difficult to carry out with Nexus 5.
- **Raspberry Pi 1:** A tiny computer, widely popular for Internet of Things deserves to be used as a compute unit. The device has 700 Mhz single-core CPU with just 256 MB of memory and also have USB support. Known to be slow in nature, may not be suitable for all class of spectrum sensing devices in our experiments. This may be the slowest device in our work but exploiting it efficiently for spectrum sensing purpose in TV Whitespace and comparing the performance with that of a high-end smartphone can help understand the need and performance requirements of mobile spectrum sensing in a better manner.
- **Beaglebone Black:** Another tiny computer like Raspberry PI but with a better performance. This device has 512MB DDR3 RAM along wiht 4GB 8-bit eMMC on-board flash storage. The Processor present on this device is AA3358 ARM Cortex-A8 with clock rate of 1GHz which makes it suitable to use for even software

[1]The USB OTG support is a must requirement for mobile device to support spectrum sensing as we can attach the USB powered sensing unit with the mobile device over this interface only

CPU: Quad-core 1.9GHz ARM
GPU: Adreno 320
OS: Android
Memory: 16/32/64 GB
RAM: 2GB
Power: 2600 mAh Battery, USB

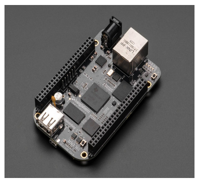
(A) Samsung Galaxy S4

CPU: Quad-core 2.3GHz ARM
GPU: Adreno 320
OS: Android
Memory: 16GB
RAM: 2GB
Power: USB, Non Removable 2300mAh

(B) Nexus 5

CPU: Single core 700 MHz ARM
OS: Raspbian
Memory: 16GB sdcard
RAM: 512MB
Power: USB, 5V Power Supply

(C) Raspberry PI



CPU: Single core 1 GHz ARM
OS: Overwrite with Ubuntu/Debian
Memory: 64GB sdcard, 4GB Flash
RAM: 512MB
Power: USB, 5V Power Supply

(D) Beaglebone Black

FIGURE 3.3: Various Hardware platforms used as Compute Units

radios and applications that have high processing requirements and it was not feasible to exploit them on Raspberry PI 1 for real time scenarios.

Before moving further, we need to define spectrum occupancy. Spectrum occupancy can be defined as the task of finding out how much occupied is the spectrum - whether a particular channel in a particular spectrum/band (TV band here) is being used by primary or it is available to use as Whitespace for secondary devices. There are many possible algorithms to find out the spectrum occupancy. All of them do not perform well in all situations. Also, computation in a low-power sensing environment is another challenging task. Below, we have briefly described the algorithms and in next chapter, we look at the performance of all these algorithms on actual compute units so as to make a real case. Different algorithms used for spectrum occupancy are described below. More details can be seen in [31].

- **Energy Based Detection:** Calculate total energy contained in the TV channel, if the total energy is more than the noise floor [2] of the device, report it as the presence of TV signal else there is a Whitespace. Total energy in a TV channel can be

[2]Each sensing device has a noise floor that needs to be calibrated which can be performed offline by operating the device in a known white space channel known beforehand and note the relative signal strength. This signal strength can be used as the threshold and can be stored in the device or spectrum server

calculated by processing IQ samples to calculate the Fast Fourier Transform(FFT) [3] [53] and then summing up the power spectral density of all the FFT bins in desired frequency range. Devices need to be calibrated before for the noise floor calculation. As per [31], this approach is known to detect the presence of strong to moderate TV signals but fails to detect the weak signals.

- **Feature Based Detection:** This technique looks for some known feature of the signal in the power spectral density, like detect the presence of a pilot in case of TV signal (In ATSC signal, it is located at about 309 KHz away from the lower frequency edge). The pilot signal power is known to be about 11.3 dB less than the total power contained in the channel. At the same time, neighboring bins of pilot tone have lower signal strength than the pilot signal itself. We can compute the FFT in a lower bandwidth (around nearby neighboring bins) to detect the pilot and hence the Whitespace. Clearly, more number of points leads to detect moderately weak signal as well which was not the case with energy-based detection. [31] provide more details into this, while [17] and [15] discuss similar techniques to track the pilot in the presence of high noise floor.

- **Autocorrelation Based Detection:** Autocorrelation of a signal is the correlation with itself. If the input signal is periodic with some period t and has multiple cycles within the scan period (time taken to scan, say N IQ samples), then the Autocorrelation(ACF) function also show the maximum correlation at period p. The pilot of a TV signal is a sinusoid signal and autocorrelating the samples obtained from the bandwidth containing this pilot also have a sinusoid along with other signals. This technique is known to detect very weak and noisy signals, unlike above two techniques. Moreover, the FFT of autocorrelation only show the frequencies observed in ACF and presence of desired frequency can easily help us determine the channel occupancy.

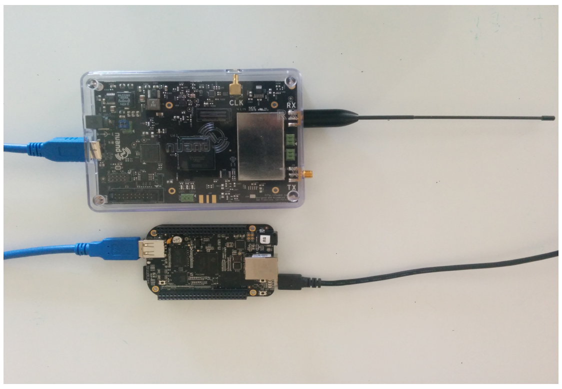
3.1.3 Communication Interface

Communication Interface is the entity binding together all other units of this mobile sensor prototype. This interface can be used to talk to the central spectrum server in distributed mobile sensing environment. The interface is also responsible for providing instructions to sensing unit, as obtained from the spectrum server and pass results obtained from the compute unit to the spectrum server again. This interface can talk to

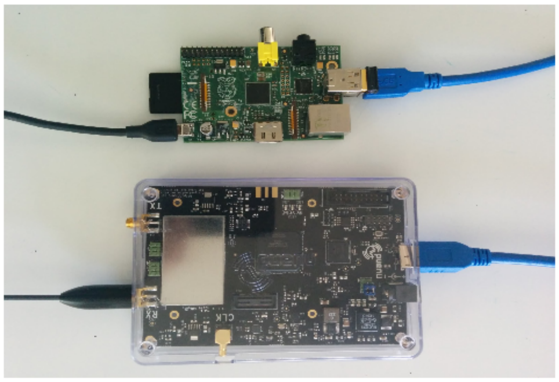
[3]FFT is a major algorithm to convert the time domain signal to the frequency domain. By processing time domain values (IQ samples) into frequency domain, we can get the power present in individual frequency bins

the server over a pre-defined protocol with a pre-defined set of APIs. The interface can also utilize the in-band Whitespace channel as well as the regular cellular/Wifi interface.

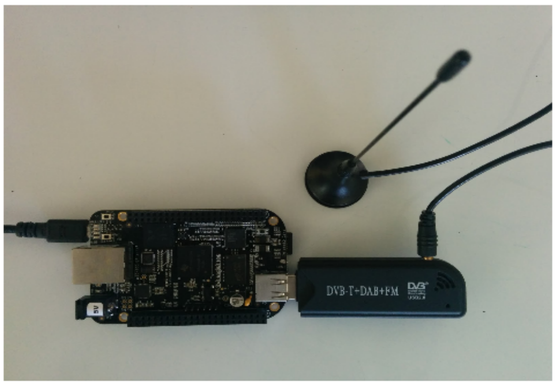
Figure 3.4 show a real picture of different mobile spectrum sensor prototypes used over the course of this work by combining different sensing and compute unit discussed above.



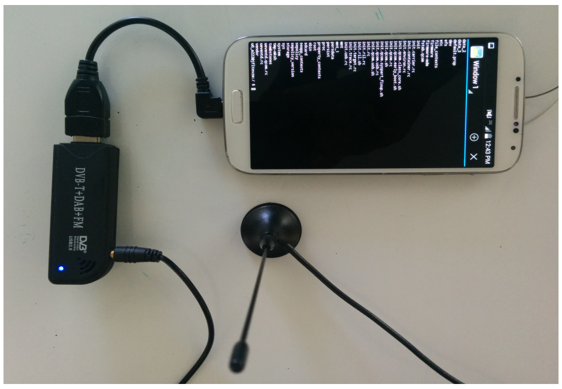
(A) BladeRF with Beaglebone Black


(B) BladeRF with Raspberry PI

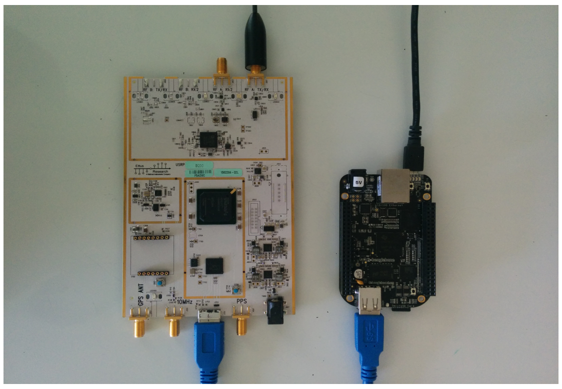
(C) RTL-SDR with Raspberry PI

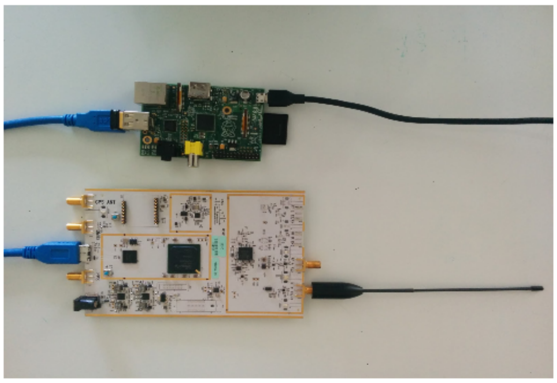


(D) RTL-SDR with Beaglebone Black



(E) RTL-SDR with Samsung Galaxy S4


(F) USRP B200 with Beaglebone Black


(G) USRP B200 with Raspberry PI

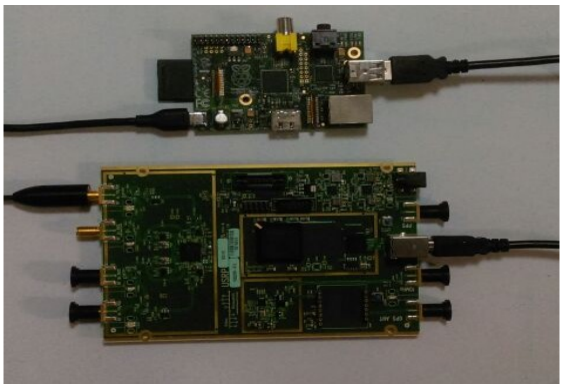

(H) USRP B210 with Raspberry PI

FIGURE 3.4: Realization of an actual Mobile Spectrum Sensor

Chapter 4

Performance Evaluation

In this chapter, we evaluate the performance of the mobile spectrum sensor in general with our prototype testbed for a real spectrum sensing environment. By performance, we here refers to the resource consumption by individual units of the mobile spectrum sensor. We discuss our chosen metrics of interest and then proceed to a detailed overview of our benchmarking setups for the performance evaluation of sensing and compute units and finally wrap up with a detailed discussion of our observations and results obtained from the experiments.

4.1 Performance Metrics

We choose two metrics of interest here for evaluating the performance of sensing and compute units - Latency and Energy.

- **Latency:** Latency defines the responsiveness of the system. A thorough evaluation of all the events occurred from the first sensing instruction passed till the spectrum occupancy decision is required to comment on the responsiveness of the system. All these events may involve but not limited to - switching on the sensing device, retuning the sensing unit parameters to a different configuration, calculate power spectral density in compute unit by processing IQ samples(received from the sensing unit) and finally making a decision about the spectrum occupancy. All these events may contribute individually to the overall responsiveness of the system.
- **Energy:** Although all the sensing units are low-power devices, they vary in cost and provisioning. All these SDRs may also have different power requirements in

various configurations of the mobile spectrum sensor. Similarly, all the sensing algorithms may also consume different energy but vary in accuracy. Thus, understanding the power/energy requirements of both the units of mobile spectrum sensor is required which can further help in making optimal choices regarding the distribution of sensing tasks among mobile spectrum sensors in general based on available/remaining battery profile.

4.2 Technical Challenges/Design choices

In this section, we have listed below design challenges, implementation choices and steps taken to simplify the setup, we need to understand this before we can proceed to understand the actual benchmarking setup.

1. We carry out the experiments separately for sensing and compute units for two reasons. First, it gives us a fair idea about which unit is more costly and what design choices we should make to deal with them. Further, power measuring setup is not easy going. Adding one more unit can make the task even more difficult. Although we see in next chapter, how combining both units can be utilized to improve the performance of compute unit but we leave that discussion for later.

2. One major challenge in conducting this study was about how can we measure the power consumed by sensing units in an efficient manner. All sensing units are USB powered devices and we can not rely on software running on the host device (with which SDR is connected) to measure the USB power [1]. We need some way to break the USB connection in between and get control of the current flowing through it. For this, we can not afford to break each USB bus in SDR or the USB channel of the host device. Rather, we can make up a USB extender cable that can be used for connecting the SDR with the host device and we can break the power flow over this cable for measurement purpose. This seems like a better and cost effective way to measure the USB power. Even in case something went wrong with the wiring, we can always buy a USB extender anywhere within a cost of about 5-6$. We discuss them in detail during the actual experimental setup discussion.

3. To measure the latency values, we need to make sure that we are using correct libraries. A library can easily result in incorrect timings if there are system processes that change the timer (like NTP). This can easily make the timer jump forward and backward in time. As a safe side, Wifi and Internet connectivity are

[1]There are many components running in the system; we are interested only in knowing the power consumed by sensing device over the USB bus, not the total power by any software on host platform

also kept off during the timings experiments. A library like "clock_gettime()" can also be used that suffers from fewer issues due to things like multi-core systems and external clock settings, although not all of these libraries are available on all latest versions of kernel and OSes.

4. Another major question of interest here - What power measuring device to use? Although there are many cheap devices like [54] that can be used to record the USB power of sensing unit, handling of these devices is not easy and the values reported are error-prone. We need better and more accurate power measuring devices that are accepted by the community. Looks like from all the main good performance work in the past, Monsoon meter [55] is undoubtedly one of the best choice for it. Although doing experiments with this device does not come handy and need extra little care and attention. If main channel (Vcc and GND) is used for measuring USB power, we need to make sure the USB wire has been properly stripped of and we are connecting the Vcc and GND of monsoon meter with that of the Vcc and GND of USB wire. Attaching them in reverse or some other inconsistent manner (like connecting directly with the cable, not the inside copper part) may hang up the device or may leave it in an inconsistent state. Same is the case with connecting monsoon main channel with the smartphone. As the setup is somewhat complex, we also need to make sure that the two ends of wires do not touch each other else it may form a short circuit loop which may even brick the device. For smartphone, Battery Size (in our case, 2600 mAh) need to be specified correctly. A proper voltage needs to be supplied (On the smartphone, 4.2 V is the suggested, this much phone take when it is fully charged. Although it also works on 3.8V and 3.5V as well, which generally refers to low-power states of phone when it does not have enough battery). [56] provides useful details about using Monsoon Meter in the proper manner.

5. To measure the power over the smartphone is a more challenging task. It is suggested to do experiment with a device having a removable battery (like Samsung Galaxy S4) than with non-removable Battery (like Nexus 5). A little problem may brick up the device completely, so we choose Samsung Galaxy S4 for our experiments. Further, for boot purposes, battery need to remain connected. As soon as phone boots up, we can remove the battery immediately (Monsoon meter supply the power to the phone). We also need to make sure that monsoon meter is not going into any calibration error.

6. There is no need to use the main channel of monsoon meter always as it adds extra complexity. For a device like Raspberry PI, it is advisable to use the USB channel of instead as it is comparatively easy going and straight forward.

7. At the very heart of our algorithms is FFT, which is used to process IQ samples and calculate power spectral density. We need to make sure we are using an efficient implementation of FFT algorithm. FFTW [57] is an improved and modified implementation of FFT algorithm, making use of novel code-generation and runtime self-optimization techniques along with many other tricks that make the library a suitable candidate for our experiments.

8. There are also system memory limitations that may not allow to move beyond a certain stack size, so it is better to optimize the code by keeping IQ sample arrays in the global static area of C code.

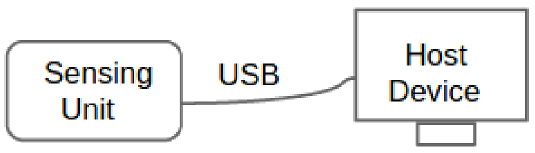
9. Making use of threads for callback routines to get data from the sensing unit makes more sense and may count for a good optimization, otherwise it may result in blocking call overhead that can further add extra latency overhead.

4.3 Experimental Setup

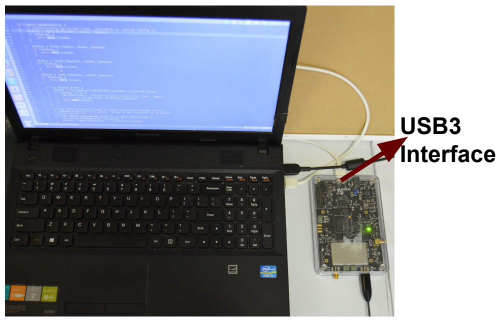
As discussed in last section, we are interested in the measurements and analysis of latency and energy consumption by individual units of our mobile spectrum sensor. In this section, we discuss all of our benchmarking setups in detail for each of such experiments. Although we use all four SDRs as sensing units in our experiments (because we want to compare the evaluation metrics between all these heterogeneous devices), we discuss our observations and setups using only two compute devices for brevity - Samsung Galaxy S4 and Raspberry PI 1. While Samsung Galaxy S4 is very well provisioned in hardware, Raspberry PI 1 is a comparatively slow device. These two devices marks the different ends of our computing platforms - Galaxy S4 being the fastest one and Raspberry Pi being the other end. The performance of other two devices can be assumed to fall in between. Thus, an evaluation of performance metrics on only two compute platforms is sufficient.

4.3.1 Sensing Unit Latency

We conduct a measurement study to analyze and compare the SDR devices for latency requirements here. The experimental setup for this section is fairly straightforward. Figure 4.1 shows a block diagram depicting this scenario. It consists of a host device (Laptop with Linux platform) attached to a sensing unit (Software defined radio under measurement). We used a Linux Laptop (4 GB RAM, Core i5, USB 2/3) running GNURadio 3.7 as a host device. The host device should have enough processing and memory requirements so as to run GNURadio. Note that all SDRs are USB powered and



(A) Block Diagram



(B) A Real Picture

FIGURE 4.1: Experimental Setup: Sensing Unit Latency Calculations

it is best to utilize the USB3 channel, whenever available. We modified the driver codes for these software radios as per the needs of our system. For RTL-SDR, we modified the rtlsdr package (librtlsdr.c and rtl_sdr.c). For USRPs (both B200 and B210), we modified the python spectrum sensing code and for BladeRF, we modified and compiled the various files in the bladerf-cli application, all distributed as part of GNURadio subsystem.

For every experiment, we instruct the SDR tuner to tune to different frequencies in the steps of varying size (from a minimum step size of 100KHz to 100MHz) and we did it for a long run of varying number of samples (from 10K to 1000M) along with different sample rate configurations (from minimum supported to maximum as per the sensing device capability). For retuning experiments, we also changed the sleep time in the steps of 1 ms to 100ms before tuning to next frequency.

To measure the latency, we noted the time before and after the benchmark point and calculated the difference. (For example, passing instruction to the tuner to change the frequency in case of RTL-SDR). We note down the time spend by individual SDRs in various states (defined in next section) and reported the results taking the average value. Figure 4.1 also shows an actual setup involving BladeRF with the laptop.

4.3.2 Sensing Unit Energy

In the previous section, we looked at various latency requirements of these sensing units. In this section, we try to understand the power needs of each sensing unit which is a major question since the beginning of our work. All the sensing units are USB powered devices connected with low-power compute units and high energy overhead (if exists) by these sensing devices can affect normal operation of a compute unit and may even force them to go out of battery. For a normal realistic operation, the power consumption

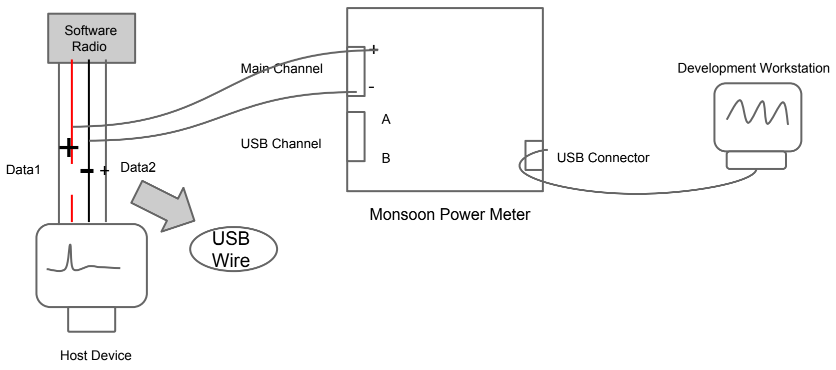


Figure 4.2: Block Diagram: Sensing Unit Energy Calculations

of these sensing devices should be comparable with the power consumed by some daily day-to-day application running on the compute platforms.

The experimental setup for this section consists of a software radio (one of each four SDRs), a host device (Linux platform), a development workstation(Windows platform) and a power measuring device (Monsoon Meter). A computer is chosen instead of mobile device as host for the convenience of the experiment; it has no impact on the measurements/results as the sensing unit is externally powered by the monsoon power meter. Also, it saves from the unwanted computational bottlenecks that may have arose if the mobile devices are used as the host device. Software radio as discussed before (See Technical Challenges), can not be attached to the host device with the help of a USB cable. We need to use the USB extender cable for this experiment. Note that BladeRF runs over the USB3 interface, so we create two set of special USB extender cables (one for USB2 and one for USB3) for this experiment. BladeRF and USRP B200 works on USB3 A type while USRP B210 works on USB3 B type, and RTL-SDR supports only USB2, so we need to create three extra USB cables accordingly along with these two USB extender cables. No SDR is directly connected but with the USB extender cables. The measurements have been done in the units of seconds with a variety of ticks and units/ticks options depending on the granularity required.

To measure the power flowing over USB cable, we remove the outer covering of USB wire so as to expose the USB Vcc(+5V) and GND wires. We disconnect the Vcc connection with the host device by cutting the red link (as shown in figure 4.2), although GND must be same for all the devices so it should remain connected. The host device is the same Linux laptop as used for latency experiments running all the respective driver codes for all SDRs. The development workstation is a Windows laptop, capable of running

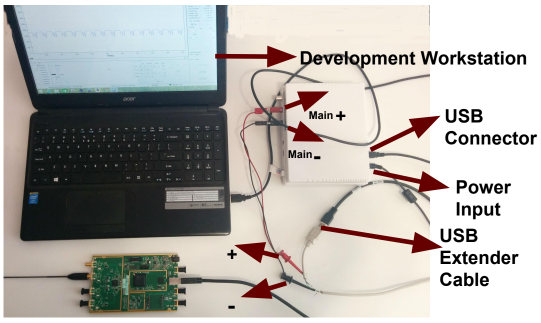
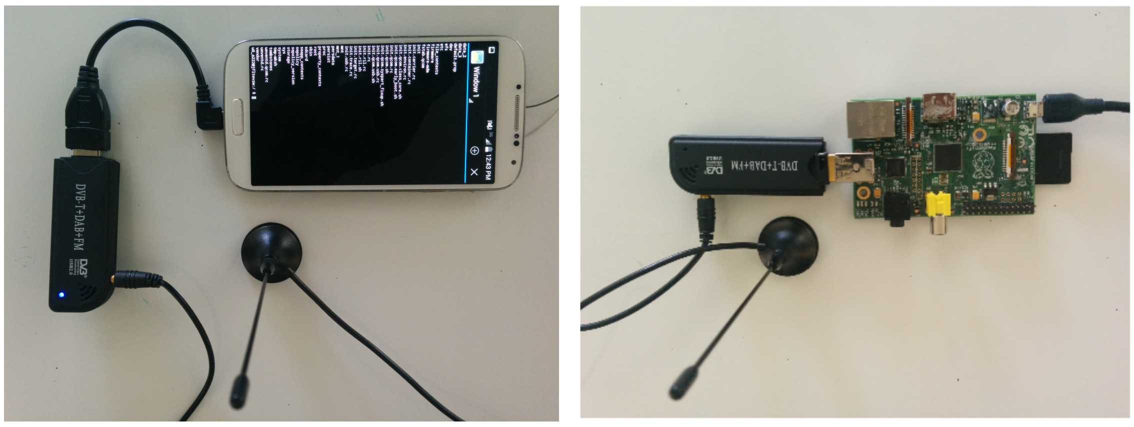


FIGURE 4.3: Real Picture: Sensing Unit Energy Experiments

PowerTool Software. We connect the Vcc(+) terminal of monsoon meter to one end of USB Vcc (coming from the software radio side) with the help of a red wire (as shown in figure) and similarly connect the GND terminal of monsoon meter with the USB GND. This kind of setup makes it possible for the actual data coming from the radio device to divert towards the host device while USB power can be measured with the help of monsoon meter. Development workstation logs the values obtained from Monsoon Meter via the USB connector. We log the results into csv files for further relevant processing.

Figure 4.2 shows the block diagram depicting this scenario and figure 4.3 shows the setup using USRP B210 with a real picture. Note the two cables behind the monsoon meter, one goes to the power plug, another one is USB connector cable connecting back to the development workstation shown in the picture. Also, look at the USB extender cable, USRP B210 connects to this cable which goes further to the Linux based host device (not shown in picture).

4.3.3 Compute Unit Latency



(A) Galaxy S4 Latency Experiment (B) Raspberry PI Latency Experiment

FIGURE 4.4: Real Picture: Compute Unit Latency Experiments

The Experimental setup for this section does not have any added complexity like previous section. We just need to run the algorithms on compute unit platforms and measure the execution time in a variety of configurations. For comparing sensing algorithms, we choose two platforms - One of the Android Phones (Samsung Galaxy S4) and Raspberry PI 1 (Model B). We looked into latency usage of all three algorithms for all these devices by changing number of FFT bins and sensing time. Figure 4.4 shows real pictures depicting the behavior of Samsung phone and Raspberry PI attached with RTL-SDR dongle. For logging latency values, we develop native Android NDK applications and record values for different configurations respectively. These individual applications run each of the sensing algorithms in a continuous loop (each iteration includes computation only) over the data captured from the sensing unit. We recorded the respective latency values for each iteration and took an average of latency values observed over more than 1000 iterations for each algorithm. We cross-compiled the FFTW library for Android platform and developed C code for all three algorithms using this. We also cross-compiled the RTL-SDR package and libusb driver for the android so as to be able to run the dongle and get IQ samples on the phone. A Similar process was repeated for the Raspberry PI. To emulate the sensing time, we divided the total number of samples (1M) by an averaging factor. The process to calculate averaging factor is as follows - Suppose we are reading 1M samples with a sample rate of 1M, it means we are reading 1 million samples in 1 second. If we want to keep the sensing time 100ms, then we want to keep only IQ samples available within 100ms. In this case, desired IQ samples becomes $1M/10 = 100K$ samples. Here, averaging factor becomes 10 and number of desired samples become 100K. We can supply these IQ samples to each of the algorithms for further processing. We recorded latency values in two scenarios. First, we note the total computation time, which includes averaging the samples as well as computing FFT on it. Second, we note time only to compute the FFT, which becomes independent of the sensing time. For different sensing times (from 1ms to 1s), we also changed the number of FFT bins (from 64 to 2048 in our case). Note that there is no actual need to attach the dongle to the compute unit for this setup. As we are only interested in the computation overhead, we can emulate IQ samples as an in-memory data structure and do further processing on it only.

4.3.4 Compute Unit Energy

The experimental setup for this section is going to be by far the most complex. Figure 4.5 and figure 4.6 describe a block diagram showing the working. We discuss first the working of smartphone and then we briefly discuss the differences with the Raspberry PI.

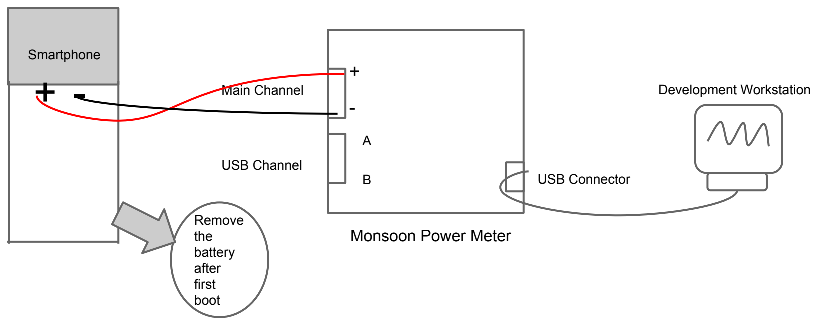


FIGURE 4.5: Block diagram: Smartphone Energy Measurements

For smartphone, setup consists of a smartphone (Galaxy S4 as we need to remove the battery for measuring power), Monsoon Power Monitor and a development workstation running Windows and PowerTool as discussed in sensing unit energy calculations. The major difference here is that we need to supply power to the smartphone to boot it up and run our android code. We can not supply power using the battery because in that case we get the wrong power values by Monsoon Meter. Monsoon Meter should act as a power input source in this case [2]. We also found that it is necessary to plug in the battery for initial boot purpose but once it is connected with PowerTool software on the development workstation, it is fine to work without the battery and we can move forward with actual experiments [3].

Also, we could not be successful in using the USB channel for this case (if we remove the battery, it gets switched off). Probably the reason might be that when we supply power using USB, it needs the battery in place. Note that this is not the case with Nexus 5 as the battery is always in place but we may get wrong values reported by PowerTool. (We figure it out by looking into PowerTool which shows Calibration Failed Warning in this case). So, we used the main channel for our work. The phone battery has four terminal in our case (Some phone has three only). We need to connect the +Vcc and GND terminals of the phone with +Ve and GND of Monsoon Meter. For initial boot up, we need to keep the battery and as soon as it boot sup, we can remove the battery and see that it shows as fully charged if we supply 4.2 V by the monsoon meter on Vout (If we supply 3.5-3.8 V, it may show low battery warning that may be misleading). Now

[2]Phone is operated by battery but to record the energy, battery can't remain plugged in as power input source, it provides the DC power to phone and we won't be able to record the actual power values

[3]We do not necessarily need to remove the battery; it can be kept in place by just blocking the +Vcc and GND of the smartphone with the help of some insulating material

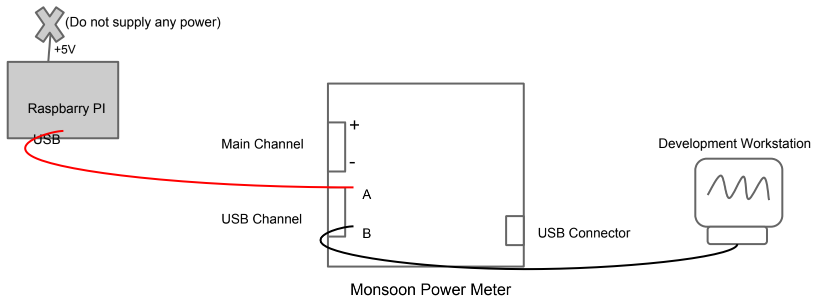


FIGURE 4.6: Block diagram: Raspberry PI Energy Measurements

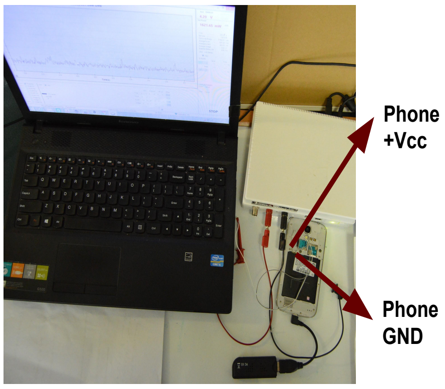
as we have got access to PowerTool, we can switch on the terminal emulator application (or any other terminal application) on Android and run our code. Note that it takes some time for the phone to come into the idle state. We can look into the power tool that initially when boots up, it takes much power, but slowly it tries to settle down. Also, we make sure that all other major applications (especially WiFi is turned off, else the values may be wrong).

For Raspberry PI, we have an added advantage that it does not come with any added battery, rather it operates on the 5V power supply and USB only. We can use the USB channel of Monsoon Meter in this case (so no added complexity of copper wires like before) but make sure that we do not supply the 5V power supply else results can be misleading. The rest of the working is similar to measuring the energy for sensing unit, like logging csv values for plotting purpose, etc. Note that we need to use a powered USB hub to successfully use the RTL-SDR or any other USB powered sensing unit with these low-power compute platforms, although as discussed before it is completely unnecessary to use RTL-SDR dongle for this experiment. Figure 4.7 shows a real picture of this setup.

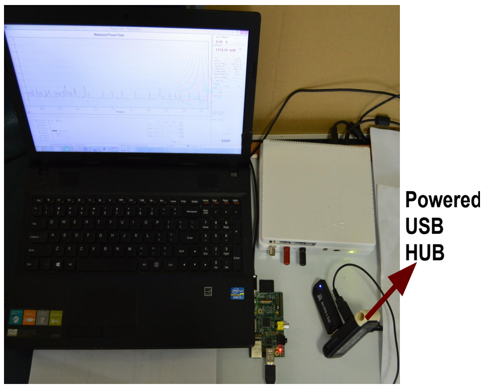
4.4 Discussions

4.4.1 Sensing Unit

We have seen all of our technical setups in detail in previous sections. Now, we discuss our findings and results here.



(A) Smartphone Energy Measurements



(B) Raspberry PI Energy Measurements

FIGURE 4.7: Real Picture: Compute Unit Energy Measurements

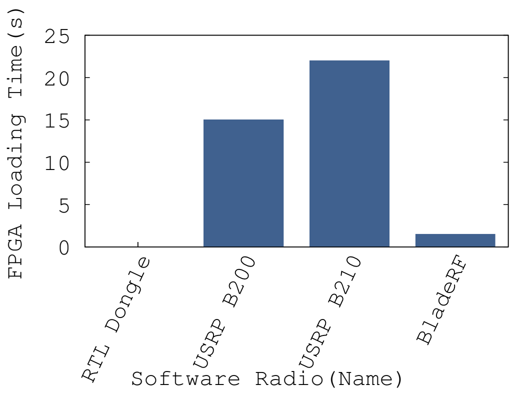
1. We discussed that different sensing units might exhibit different latencies that can define the responsiveness of the system. We look at the individual latency exhibited by each SDR in various configurations here. We first define and discuss different latencies below and then discuss the results.

   Definitions:

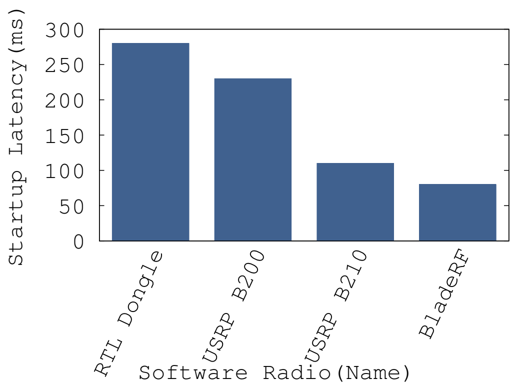
   - **FPGA Load Time:** We can divide the four SDRs into two different class of devices - FPGA class of devices and Non-FPGA class of devices. While BladeRF and USRPs (both B200 and B210) are FPGA class of device, RTL-SDR does not use any FPGA in particular for its operations. FPGA class of devices also loads the appropriate firmware before operating. In our experiments, we measure the latency consumed by sensing units for FPGA loading. A high FPGA load time may be exhibited by devices having more features due to the added complexity.
   - **Startup Latency:** It can be defined as the time elapsed when the sensing unit gets the first instruction to start sensing with a set of parameters (e.g., sample rate, number of FFT bins, sensing time, center frequency, number of samples etc.) and when it actually starts getting the samples. The startup delay is inherent to the hardware. For example, we verified and found that RTL dongle turns on the tuner chip during this time that separately consumes about 45ms time while demodulator consumes about 65 ms.
   - **Retuning Latency:** When the sensing device is up and running, cloud-based spectrum server may instruct the device to scan with a different value for some parameters. For example, the sensor may need to scan in a new center frequency. The amount of time it takes to start sensing in this new frequency is referred as the retuning latency.

- **Shutdown Latency:** There is one more aspect of latency of software defined radio, the shutdown latency. This is the time elapsed when the device is instructed to stop the sensing and when it actually stops completely (without getting turned off). This may not be good if sensing device needs to be turned on and off frequently (to save power, for example).

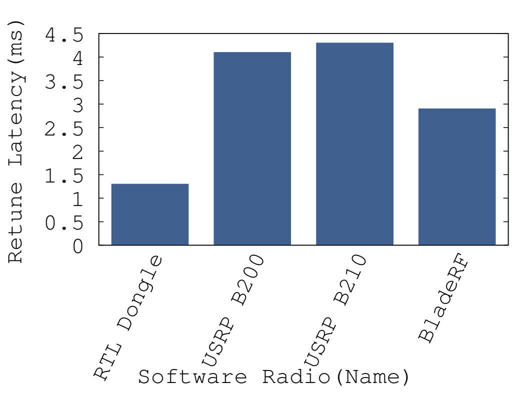
Figure 4.8 shows the comparison between devices for different latency components.



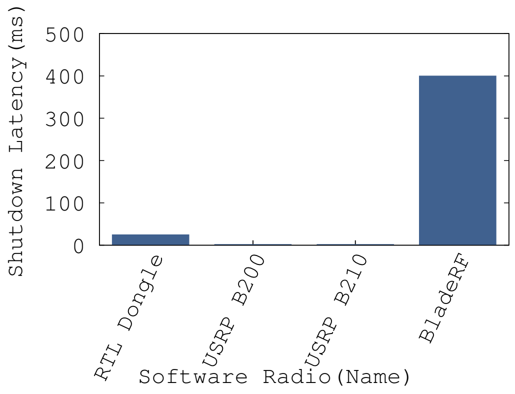
(A) FPGA Load Time



(B) Startup Delay



(C) Retuning Delay



(D) Shutdown Latency

FIGURE 4.8: Different Latencies of Sensing units

Results:

- FPGA loading time is higher for USRPs compared to BladeRF, which was expected as Xilinx chips used by USRPs have more features than Altera chip of BladeRF. Also, USRP B210 is known to be more accurate compared to USRP B200 so it may have extra components in its FPGA that may be the possible reason of a slightly greater FPGA load time. Obviously, RTL-SDR has no FPGA load time
- Startup latency exhibited by RTL-SDR is highest and slightly more than the USRP B200. This is because the RTL-SDR device is made up of only raw

hardware components without any extra optimization added by using FPGA, firmware, etc and each component takes time to switch on which adds to the startup latency. This time is reduced by moving some of these hardware component functionalities in the FPGA in case of other devices. USRP B210 show less startup latency compared to USRP B200, which indicates that the improved FPGA in USRP B210 have some extra functionality that may be available as a raw hardware component in USRP B200. Also, BladeRF has the least startup delay that indicates that the device may have less hardware and features(for example, support for a lesser range of radio spectrum) compared to USRPs. It also seems that BladeRF may be a preferred choice over USRPs if exact needs are known in advance.

   - Retuning is one of the most important part of mobile spectrum sensing. As discussed before, It defines the time difference when the server instructs the device to switch to a different channel or scan in a different frequency and the actual time when the device is tuned to new frequency. A high retuning delay can simply not serve the purpose. Luckily, numbers shown in the figure are not very high. Still, USRPs have high retuning time compared to BladeRF and RTL-SDR, which again make a point that if there is a requirement to sense the small range of spectrum by one device, BladeRF is probably better (It does not mean BladeRF is always be a preferred device). RTL-SDR, being a very cheap device can not match the capabilities of BladeRF but having many of them and each working on different part of the spectrum at a time may still make a good choice.
   - Here BladeRF is going worse compared to other devices. USRP does not show any such latencies. Now, here it becomes a trade-off between the startup and shutdown latency based upon the actual needs of spectrum sensing.

   Although we provided a comparison between various sensing units above, in general, all the latencies exhibited by different sensing units seems very reasonable (few milliseconds) and can be believed to work in a real scenario.

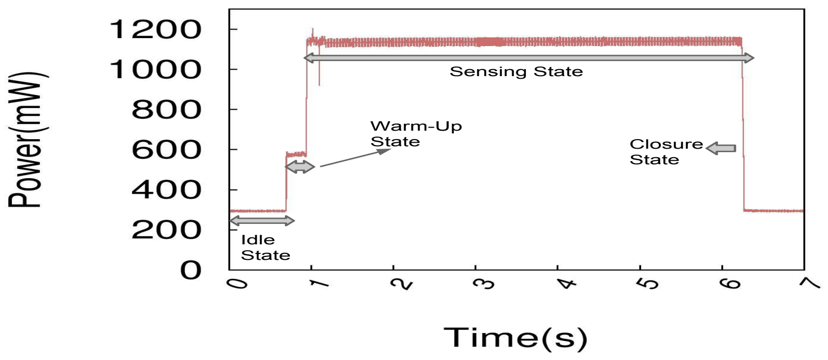
2. From our run time power profiling experiments for sensing units, we can define four possible states for any particular configuration of any software defined radio or sensing unit in general.

   - **Idle State:** The idle state refers to the SDR configuration when it is powered up but not actually sensing the spectrum. As we see earlier that the startup delay can be high for some devices so if a device senses the spectrum frequently, it makes more sense to keep the sensing unit on rather than turning it off completely.

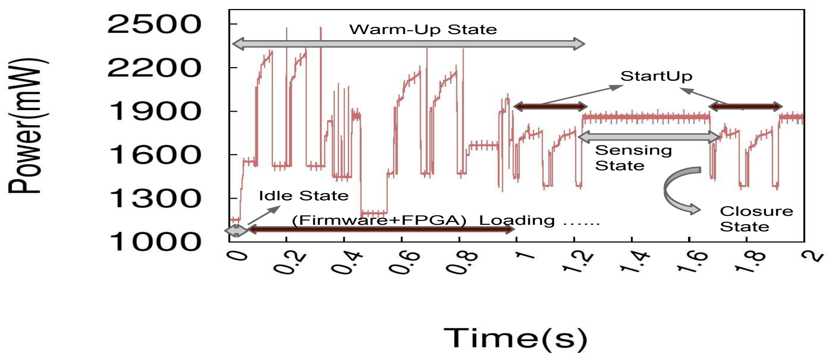
- **Warm-Up State:** The warm-up state refers to the time period between switching from idle state to sensing state.
- **Sensing State:** The sensing state can be defined as a configuration when SDR has begun executing its first instruction. The whole circuitry of radio is up now, and it is sensing the spectrum. In this state, sensing unit keeps on collecting IQ samples and continuously send them forward to the compute unit based on the instruction received from the spectrum server along with the required set of parameters.
- **Closure State:** It refers to a state in which device is going from sensing state back to the idle state. Similar to Wifi, some software radio may also involve a tail energy in a closure state. The tail energy is the energy wasted after completion of data transfer in high power state. For example, in case of Wi-Fi about 60% of energy gets wasted in this state [43]. We need to see if the sensing unit also exhibits a similar behavior and if yes, we need to understand the hindrance and impact of such behavior.

Figure 4.9 shows a time-line diagram for various power states of each software defined radio. If we look closely at the power state time-line plots for all three devices, we can see that RTL-SDR (Non-FPGA class of device) goes from idle to warm-up to actual sensing like others. We dig more into this and found that two major DSP chips define the working of RTL-SDR- The Demodulator chip and the Tuner chip. The Demodulator in RTL-SDR is RTL2832U, which is a high-performance DVB-T demodulator supporting USB 2.0 interface and also embedded with an advanced ADC. More details about this dongle can be found [58]. Now during the Warm-Up state, only the tuner gets up and consumes some power. After demodulator comes into the picture, the device goes into sensing state and start scanning the spectrum.

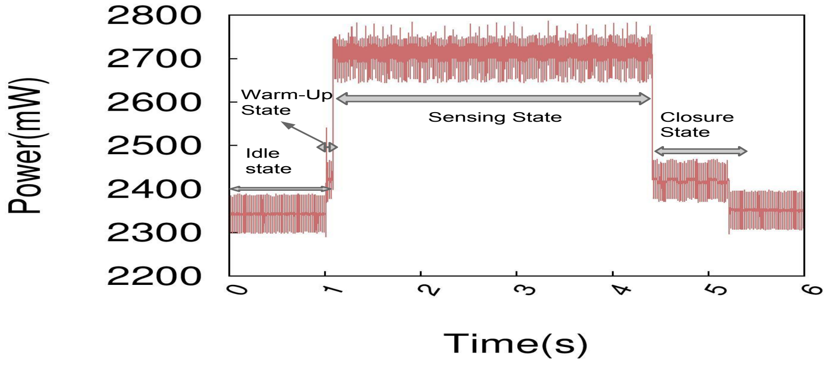
On the other end, FPGA class of devices has a different kind of working style. They needs to be loaded with a firmware that can support the required FPGA. For scanning the spectrum, driver on host machine need to pass an instruction to the FPGA. Compared to RTL-SDR, things are black-box here inside the FPGA. Initially when we get a new device of such kind, it needs to be loaded with a firmware, that is the only one-time operation. After this, when we attach the device to a host device with supported driver, the device loads the FPGA. Note that this is done every-time we plug-in the device into the system. Although FPGA provides the advantage of parallel processing over traditional serial approach, the downside is that they clock slower than microcontrollers. Also, FPGA designs are developed by users who may not be highly experienced and well versed in HDL. As discussed before, USRP FPGAs is known to contain more features than BladeRF.



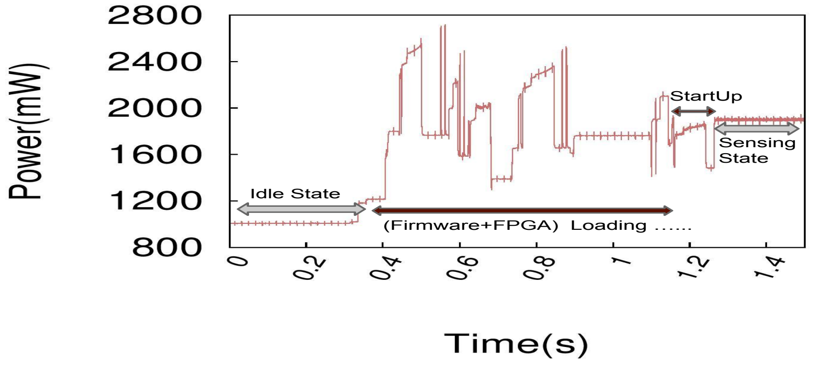
(A) RTL-SDR



(B) USRP B200



(C) BladeRF



(D) USRP B210

Figure 4.9: Different Power States of Sensing units

In the Warm-Up state, the amount of time spent by USRPs depend on whether it is very first time, it is being used (Firmware loads), are we reconnecting the device stopping all previous operations (FPGA loads) and the usual startup delay needed to power up the circuitry. The idle and sensing state are evident here. Closure state is the time taken to switch off all these components and get back to the idle states, as seen in plots.

3. We saw latencies for different sensing units. Now, we look at the power drawn over USB bus by these SDRs. Figure 4.10 shows a comparison between all SDRs between power drawn in idle state and sensing state. It is quite evident that BladeRF seems to consume very high power in comparison to USRPs and RTL-SDR. In general, RTL-SDR sounds very efficient in power aspect. USRPs consumes less power compared to BladeRF but exhibits high startup latency. Along with a better FPGA, USRP also seems to provide a power efficient solution. [31] provides more details for the accuracy. If we look at both the latency and power consumed by sensing unit, we can claim that USRP being the most accurate device (due to better architecture and FPGA choices) may induce more retuning and startup latencies compared to BladeRF but can be power efficient. On the other end, RTL-SDR being the cheapest and most noisy device takes less power. We can put more bladeRF units and even more RTL-SDR units to match with the fine-grained results provided by USRP. However, if we look at the latencies and power consumed by USRPs, it seems much better choice to use it compared to existing costly and bulky spectrum analyzers.

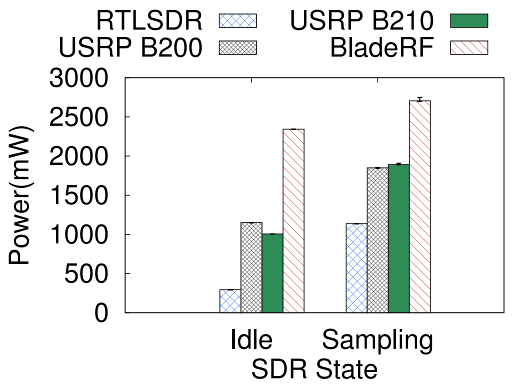


Figure 4.10: Power Consumed by Different sensing units in Idle and Sensing State

4. We also anticipated before that power consumed by any sensing unit may depend on the sample rate. USRPs being supporting the highest sample rate among all, we did experiment with it. We found that sample rate did affect the instantaneous power drawn by any particular software defined radio, but the variations are subtle compared to the base power needed to keep the device up. Also, change between

maximum (61 MHz) and default sample rate(1 MHz) is in the order of less than 5% which is not very large and thus it can be claimed that any sample rate can be used in real-time application without significantly affecting the power.

5. We also measured the power consumed by different applications on a regular smartphone so as to compare it with our sensing unit. We found that power consumed by phone with an RTL-SDR device attached (around 1600 mW) was comparable to the normal day-to-day application like YouTube (in a stable state) running on the mobile device.

   We looked at various performance aspects related to the sensing unit of our mobile spectrum sensor prototype and numbers seems to make a case for a real scenario. We hope this study can help decide in the design of sensing unit for next generation mobile devices with spectrum sensing capabilities. Now, we evaluate the performance of our compute unit.

4.4.2 Compute Unit

1. In this section, we discuss our measurement results to analyze sensing algorithms for various latency and energy cost. First, We explain and try to understand the working of a compute unit from the Figure 4.11 which shows a block diagram depicting general working nature of our spectrum sensor.

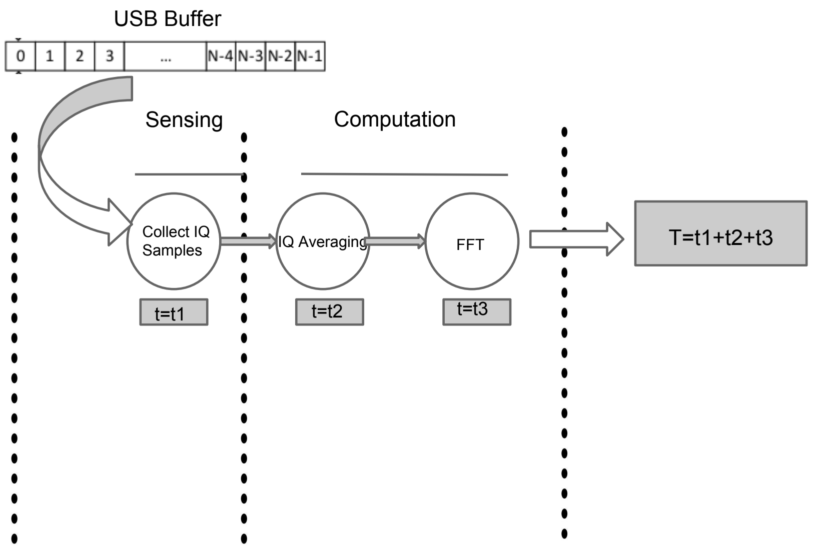
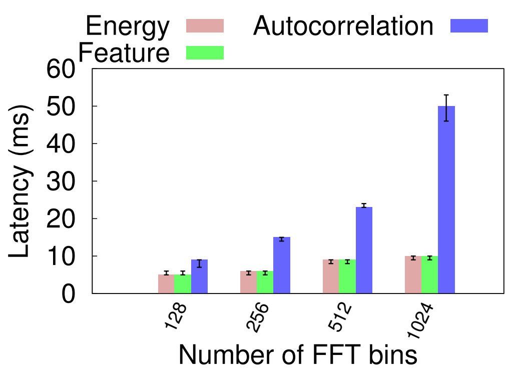
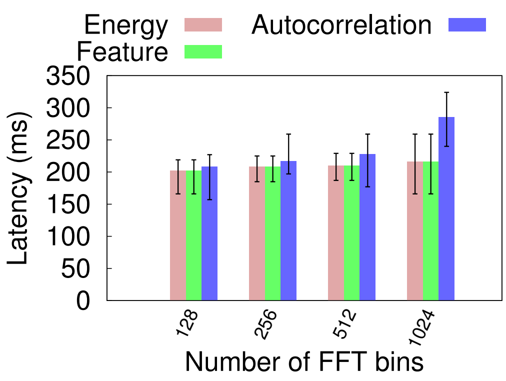


FIGURE 4.11: Block Diagram: Working of Spectrum Sensor

Working: Sensing unit is only responsible for getting data (Raw IQ samples) from the USB buffer while compute unit deals with two parts of computation - Get IQ

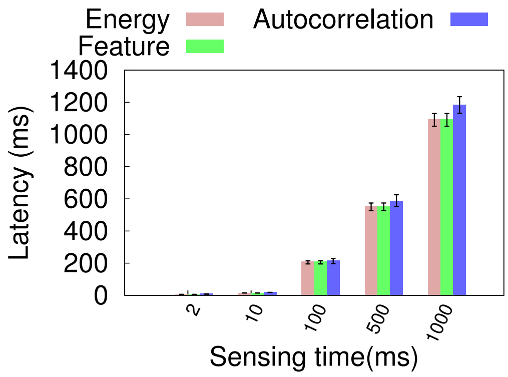


(A) T=2ms

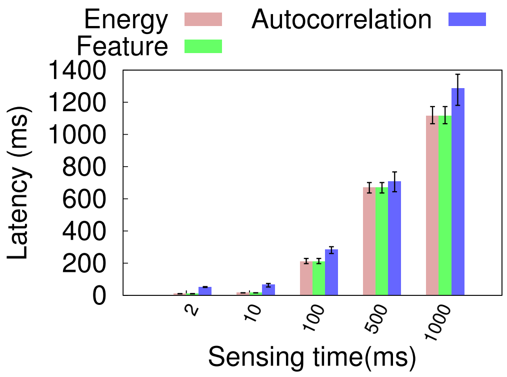


(B) T=100ms

FIGURE 4.12: Compute time(in ms) for all three algorithms varying sensing time using Samsung Galaxy S4



(A) NFFT=128



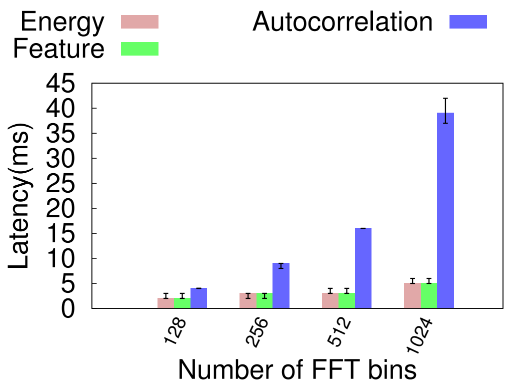
(B) NFFT=1024

FIGURE 4.13: Compute time(in ms) for all three algorithms varying number of FFT bins using Samsung Galaxy S4

samples from sensing unit to average them and compute FFT over these averaged IQ samples to calculate the power spectral density. In this Chapter, we explain one approach of implementing compute unit which gets the required number of IQ samples from sensing unit in series and then do the computation. So, total latency occurred would be total time spent in sensing unit plus total time spent in compute unit.

Figure 4.12 and 4.13 shows the results obtained for time taken by compute unit (in this case, Samsung Galaxy S4 Smartphone) to perform the task by changing sensing time and number of FFT bins. We list below our observations/findings from the results -

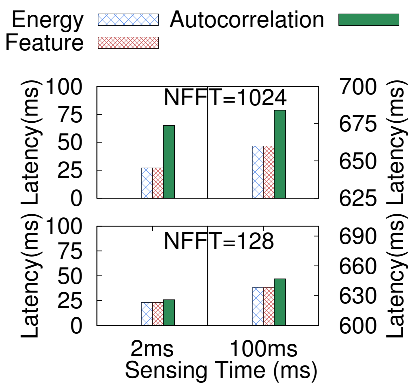
Results:



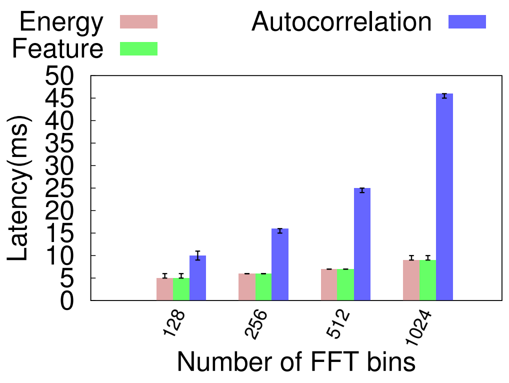
(A) Latency

FIGURE 4.14: Compute Latency for FFT (only) for all three algorithms using Samsung Galaxy S4

- We see that in general more the sensing time, more it can be computation intensive. It takes more time to finish the task. Also, Autocorrelation in general always takes more latency for computation. Energy and feature based detection techniques take same amount of time because both algorithms differs in logic only at one step after the FFT computation. In general, Energy and feature based detection techniques are not run with same number of FFT bins. For achieving similar granularity, Energy based detection need to be run with higher number of bins which takes up more time.
- Changing number of FFT bins does affect the computation but not as much as compared to the sensing time. At large sensing time, the overhead due to number of FFT bins may not even be visible well as seen in figure 4.12
- Unfortunately making things serialized (sensing followed by the averaging and FFT) has an enormous impact on the system's responsiveness. If we look at the latency values for high sensing time, the numbers seems very high (almost close to double than the sensing time) which raise a major concern about the validity of these algorithms in real scenario in the context of mobile spectrum sensing. Looking at the block diagram shown in figure 4.11, we see that computation cost involves two major tasks - IQ Averaging and FFT calculation. We need to analyze both of them independently to see which among them takes more time and look for possible optimizations.
- Figure 4.14 shows the computation time taken by FFT only and it seems that the latency values reduces drastically (Note that the latency values follow the FFT algorithm complexity $O(NlogN)$) which proves that it is IQ averaging which takes the major time. We need to look for some some techniques to improve this time.



(A) Total Computational Latency



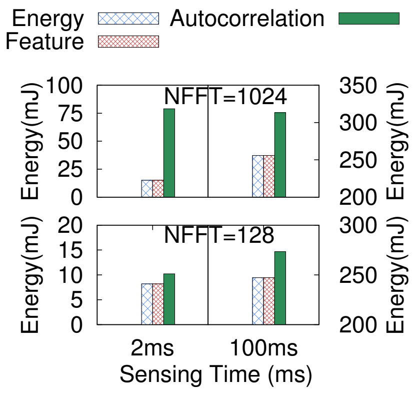
(B) FFT Latency

FIGURE 4.15: Compute Unit Total Latency and FFT Only Latency Results for all three algorithms on Raspberry PI Platform

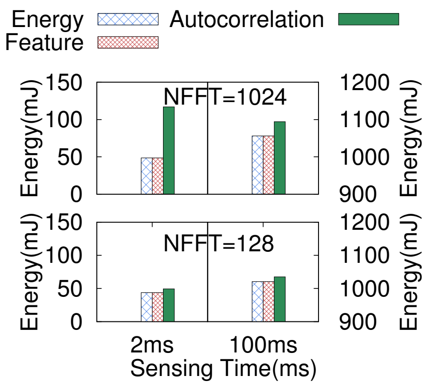
2. We repeat same experiments for Raspberry PI and results seems to follow same trends. Although there are some variations like Raspberry PI in general takes a little more computation time which was expected as it has a slow processor. Also, similar to Galaxy S4, Figure 4.15 shows the results for total latency (averaging and FFT)and FFT only for Raspberry PI 1 platform.

3. We also look into the power consumed by sensing algorithms. In general, the sensing algorithms consumes about 700-1200mW extra power compared to the base power consumed by a compute device, although we are not interested here in the power consumed rather we care about the energy consumed over a period. The remaining battery on a compute device depend on this energy consumed by the algorithms. We found that energy consumed is directly proportional to the latency values exhibited by algorithms. In general, more the latency of a particular task, more be the area under the curve (having power on Y-axis and time on X-axis and so more be the energy consumption.) Figure 4.16 show total energy consumed for running all three algorithms on Galaxy S4 device while Figure 4.15 show the same behavior on Raspberry PI 1.

4. There is another possible approach where we can upload the IQ samples on the cloud-based spectrum server and server can run the algorithms but that does not appear to be a good choice. The server can receive IQ samples from multitude of devices and running algorithms over all of them and then creating spectrum map can further induce unnecessary delays in system compared to utilizing a distributed platform on mobile device for achieving this task in parallel. Also, sending raw IQ samples to the server may not be a good choice because it introduces extra network delays as well which could be very high (in the order of seconds or minutes) compared to the processor speed. Even performing averaging on server is not a



(A) Galaxy S4



(B) Raspberry PI 1

FIGURE 4.16: Compute Unit Total Energy Results for all three algorithms on Galaxy S4 and Raspberry PI Platform

good idea as we need to upload extra IQ samples again (equal to the averaging factor) on server, creating network overhead.

In next chapter, we look at the the high computation overhead problem on large sensing time in detail and provide a mechanism to improve the performance of averaging.

Chapter 5

Improvements and Optimizations

In this chapter, we discuss the possible improvements and optimizations for the compute unit. There are few possible places that can be optimized in the compute unit. We list them below and discuss their needs and then we show actual improvements mechanism applied in further sections.

- First, we need to find some ways to improve the responsiveness of compute unit. We saw in last chapter that IQ averaging exhibit high latency values for high sensing time. Although, keeping a low sensing time can be a naive approach but we have discussed in Chapter 4 that more sensing time is equivalent to more averaging, and thus it helps in removing the uncertainty present in noisy devices so as to get better and accurate results. We see in this chapter an approach to improve the averaging latency by utilizing CPU cycles during the sensing only.
- In the next section, we also look at how we can utilize these mobile spectrum sensors in the area of wideband sensing. We see that for small number of FFT bins, FFT show reasonable computation time but for high values, these latency values start increasing exponentially and certainly not look good to be used in a real scenario.
- We further find out various major factors affecting the performance of such systems in general and try to develop an understanding of their behavior.

5.1 Block Level Averaging

5.1.1 Problem Overview

The actual problem with our initial design of compute unit is that computation can not start until the sensing task finishes. This makes us consume some extra CPU cycles which can be exploited for computation. This causes us a big computation cost overhead where for a high sensing time, like t=100ms, we have to put about extra 200ms for just the computation (on Samsung Galaxy S4). It means we are devoting almost the double time for computation only then required idle time that accounts for a less than half throughput making the problem worse. This kind of performance can hinder a real-time behavior and we need to develop some strategy to improve it further. Figure 5.1 shows the general block diagram of our improved required system. We would like to overlap a part of computation (IQ averaging) with the sensing unit in some manner which can further save us extra CPU cycles consumes for averaging later.

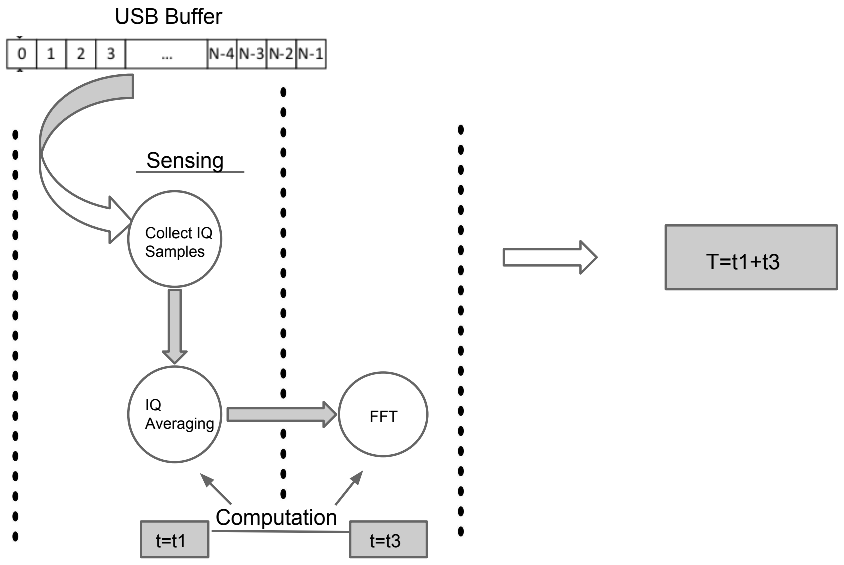
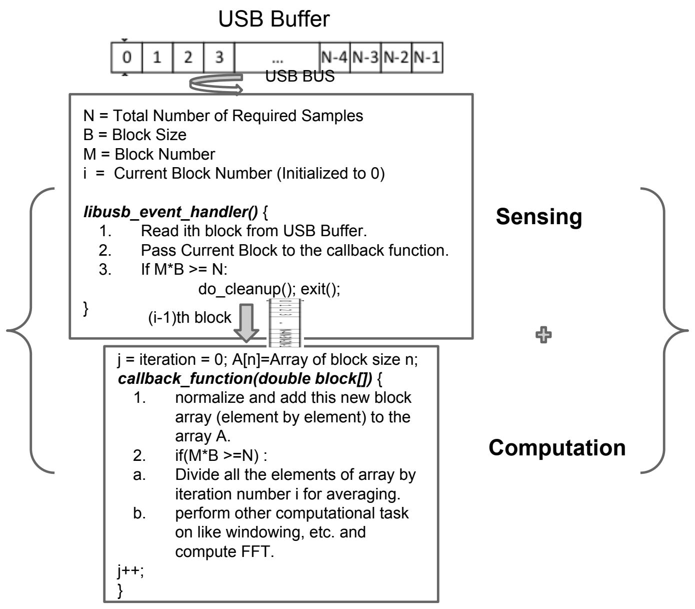


FIGURE 5.1: Improved Approach: Block Diagram

5.1.2 Solution Strategy

Here, we go one level deep inside to look at the actual working style and implementation strategy of the driver code for RTL-SDR (We utilize only one sensing platform here as we are more interested in compute unit performance, although similar approach can be applied with other SDRs as well). We found that sensing unit operates by reading data from a USB buffer in the size of blocks (block size is determined by potential users



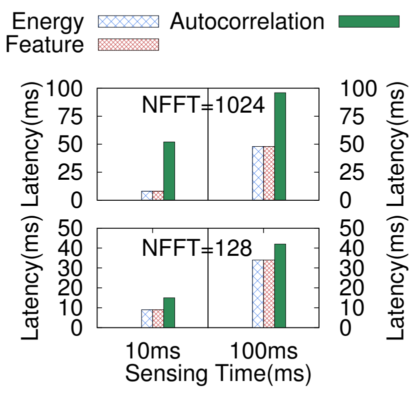
```
N = Total Number of Required Samples
B = Block Size
M = Block Number
i  =  Current Block Number (Initialized to 0)

libusb_event_handler() {
   1.    Read ith block from USB Buffer.
   2.    Pass Current Block to the callback function.
   3.    If M*B >= N:
                 do_cleanup(); exit();
}
```

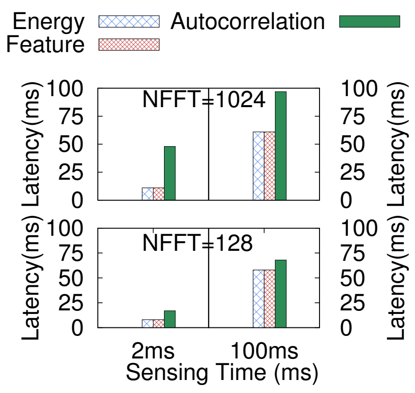
```
j = iteration = 0; A[n]=Array of block size n;
callback_function(double block[]) {
   1.    normalize and add this new block
         array (element by element) to the
         array A.
   2.    if(M*B >=N) :
   a.    Divide all the elements of array by
         iteration number i for averaging.
   b.    perform other computational task
         on like windowing, etc. and
         compute FFT.
j++;
}
```

FIGURE 5.2: Exploiting sensing and computation in parallel for an RTL dongle device in Asynchronous Manner

of the application). The sensing unit is actually a libusb event handler that reads data asynchronously from the USB buffer in terms of blocks and pass this data to the callback function which keeps accumulating this data in an extended IQ buffer of size equal to the total number of required samples(Say N). In our approach before, we used to wait until this IQ buffer gets filled completely and then we start our computation (First averaging and then FFT). We propose a slight modification to this approach. The approach says that rather than keeping a long buffer of size N, we only keep a small buffer of size equal to block size and keep track of number of times the call has been made for this callback function (Number of iterations being made till we read total N number of samples). In each iteration, we keep adding the blocks (the new one passed from the event handler and our small old buffer in the callback function) value by value in-place using the small buffer in the callback. At the last iteration thus, we only need to divide each member of this buffer by the number of iterations and we are done with the averaging. We can then compute the FFT over this buffer in one further extra step. Note that the size of this small buffer should be greater than the twice of required number of FFT bins for a valid FFT operation (discussed n next section). In this way, we can save extra CPU cycles wasted before for IQ averaging purpose before. Figure 5.2 shows the working algorithm for this setup. In one cycle, sensing and computation take place in a parallel manner.

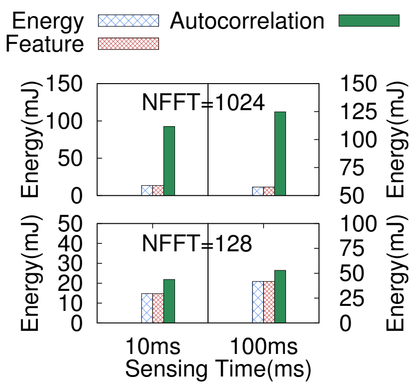


(A) Galaxy S4

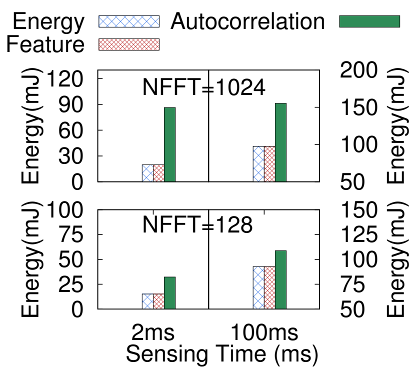


(B) Raspberry PI 1

FIGURE 5.3: Improved Compute Unit Total Latency Results for all three algorithms on Galaxy S4 and Raspberry PI Platform



(A) Galaxy S4



(B) Raspberry PI

FIGURE 5.4: Compute Unit Improved Total Energy Results for all three algorithms on Galaxy S4 and Raspberry PI Platform

5.1.3 Results

Here we discuss our improvements from the above asynchronous algorithm. We run our code in the similar configuration as discussed in previous chapters except that block size is twice the number of FFT bins [1]. We observed a drastic improvement in the computation time. The results after improvements on both devices are mostly dependent on FFT computation The averaging factor has been taken care of very well. Note that Galaxy S4 does not show any improvements till 10ms. The Reason is that Galaxy S4 was already performing very well before in previous serial algorithm (for e.g. - it takes about 5ms for running Energy based detection for 2ms sensing time) compared to the Raspberry PI 1(takes about 24 ms for running Energy based detection for about 2ms

[1]Block size should be equal, in general, here we do it twice as one FFT bin refers to one IQ sample containing 2 data samples (I and Q), while data in the block would be a raw data, so 2 data samples in a block makes 1 FFT bin

sensing time). The asynchronous calls also have a little overhead that appears below 10ms in case of Galaxy S4. Thus, using serial algorithm seems a better approach till 10ms. Figure 5.3 and 5.4 shows the improved latency and energy values for both the platforms.

5.2 Improvements using GPU

Smartphones are believed to be power hungry devices and have been considered unfit for computationally intensive tasks until recent past. Now, modern smartphones have increased not only in terms of memory and processing capabilities but these low power SOCs come with high end graphical processing units. Now, these devices offer an attractive platform to run computationally intensive software defined radio applications. In this section, we try to explore the feasibility of executing FFT code on an Android device and look for its possible implications in software radio domain. Although the idea of executing code on graphical cores looks very interesting and there are a wide variety of software stack available in market, there is still lack of software maturity when it comes to execute GPU code on ARM-based low-power devices. OpenCL probably looks like the best possible choice for executing parallel code on mobile GPUs. It has been widely popular and accepted framework for a long time in high-performance computing community. Despite wide acceptance of OpenCL, there are still many challenges involved when it comes to executing OpenCL code on mobile GPUs as mentioned here. The work in [59] present a nice overview regarding this. In next section, We discuss our methodology for using OpenCL on Android to execute the FFT code and discuss corresponding results. We also run FFT on high- end CUDA machine to compare the performance improvements over the phone with that of a high end GPU cluster and makes a case for how we can use it in the context of wideband sensing.

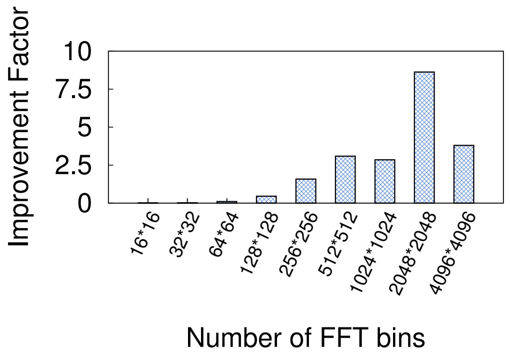
- **Wideband Sensing:** Wideband sensing [60] [61] is an emerging area where a need is there to sense a large chunk of spectrum, compared to the narrow-band sensing. For example, if we want to sense the whole TV spectrum at once (470-698MHz) which is about 228 MHz, it takes a whole lot of time using traditional narrowband sensing techniques till now. If we think about spectrum sensing out of TV spectrum, there are even existing system which require and propose the techniques to sense a big spectrum (for e.g. - 2.4 GHz wide spectrum) [33] [36] which is not possible to achieve with these narrow-band sensing techniques. Here, we discuss the applicability of mobile spectrum sensing in such systems.

5.2.1 Methodology

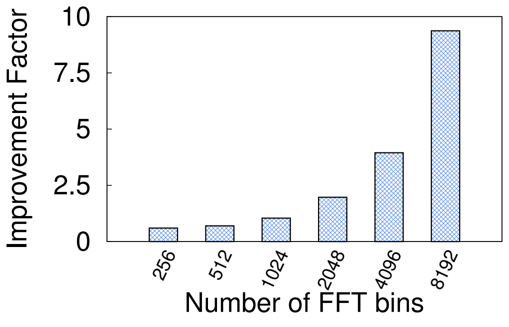
Our aim is to get an high-level overview about the possibilities of using GPU for spectrum sensing purpose in the current scenario or nearby future. In this section, we first discuss our setup for mobile phones and then we explain the same for high-end GTX-700 graphical cluster TITAN BLACK. For Android phone experiments, we use the Samsung Galaxy S4 phone having Adreno 320 GPU, that seems to be conformant of OpenCL implementations. We put the cross compiled libOpenCL.so library on Android phone and developed native C-Application using Android NDK for executing the FFT code. For FFT code, we run the Cooley-Turkey [62] as well as Stockholm's FFT Algorithm [63] for radix-2,radix-4 and radix-8 kernel by configuring minimal required set of parameters and choose the best possible results. We did not choose any optimal local work size by ourselves rather we simply choose the global work size to be equal to the size of the problem and let the compiler decide for optimal distribution among cluster work groups. Our implementation are inspired from this [64]. As mentioned before, we were interested only looking out for the possibilities of executing FFT code on GPU in general sense, our implementations may be very flimsy and not highly optimized, but they at-least works successfully on the phone. There are many optimized version of FFT code available , like FFTW, Apple FFT, etc and a lot of other tweaks are possible like auto-tuning for the best possible results. In next section, we look at the actual numbers. For Running FFT code on GeForce GTX TITAN BLACK graphic card, we choose PyCUDA and PyFFT. Also, for Android phone, the maximum possible size of FFT we were successfully able to produce results for is 8192, while for PyCUDA, it is 1024*1024.

5.2.2 Evaluation

Figure 5.5 shows the results for both the cases. While we see that for lower FFT bin size, there is no performance improvements of GPU, there is a significant boost for higher numbers. The pattern is same on the TITAN BLACK cluster although the crossover point is very away in case of later. Let us try to understand here what is going on here actually. The GPU performance on the cluster is very well accepted but how could mobile surpass the high-end cluster GPU. The answer is that it does not surpass, GPU does not provide any added advantage for higher numbers or even low numbers. GFLOPS performance is almost same as depicted here also [59] for all the numbers. It is CPU which is under-performing (taking too much time for high bin FFTs while GPU is remaining stagnant) and thus the GPU performance factor is coming out.



(A) TITAN BLACK



(B) Adreno 320

FIGURE 5.5: Performance Results for Titan Black GTX-700 Cluster and Adreno 320 GPU

5.2.3 Observations/Challenges/Design Choices

Now based on the results, we can possibly figure out some important observations -

- GPU is probably not suitable for narrowband sensing but it could prove to be very helpful in scenarios like wideband sensing. Although as we mentioned above, the performance of GPU can be improved by further optimizing the various parameters in OpenCL configuration but in general, for small number of FFT bins, FFT execute decently fast (in about 5 ms on Samsung Galaxy S4 for a 2 ms sensing). On the other end, if we look at the performance results for higher numbers, we see that for N=8192, the performance is almost 10 times of the CPU. If we evaluate it one step further, it means that if we were taking 10 ms to scan the spectrum before, now we take only 1 ms. It also means that we can scan 10 times more spectrum in same amount of time. So, If we were scanning 100 MHz before, we can scan about 1 GHz now which makes it perfectly applicable for wideband sensing purposes. People are already talking about it [50] . Recently there have been many wideband systems developed [36] [32], this kind of GPU devices may prove very helpful in these scenarios.
- Adreno 320 is not the superfast GPU, [59] it is quite evident that it is not very well on GFLOPS count and performance. There already exists much better GPUs already in market that have much better configuration and higher specifications, and that can simply be exploited to see the effect.
- As day by day, we are moving forward with technology, we can believe to get a CPU and GPU of capability like this high-end TITAN BLACK cluster on the phone in next few years and looking at the results of this cluster, this trend seems

still applicable that GPU performs well for higher bin FFTs and it can be very well suitable for wideband sensing.

5.3 Factors Affecting Spectrum Sensor Performance

In this section, we try to list some of the important factors affecting the performance of spectrum sensor in general based on our study. Some of these we have already seen in previous chapters while, for others, we present results and discuss and try to understand the reasons behind their behavior. Knowing all these parameters can help in the implementation of cloud-based spectrum server that can instruct the spectrum sensor with a right set of parameters so as to finish the sensing task with a desired accuracy but within a power and latency budget. Some of these important factors considered in this study are as follows:

- **Number of Samples:** It is the most obvious. If more number of samples are required then the server needs to instruct the spectrum sensor to scan for a long time until the desired number of IQ samples have been received by sensing unit and further processed by compute unit.
- **Sensing Time:** We have seen before in Chapter 4 that increasing the sensing time had an adverse effect on the performance of the compute unit. More Sensing time means getting more number of samples and wait for more time until we get the number of samples required to achieve the desired accuracy. Although, we see that the using the improved asynchronous callback method reduces this dependency to a great extent but still the final performance is linearly proportional to the sensing time.
- **FFT Size/ Number of FFT Bins:** Fast Fourier Transform calculates the power spectral density for individual frequency bins and is a known algorithm in spectrum sensing community. The Algorithmic complexity for FFT has been reported as O(NlogN) which means more the number of IQ samples to be processed, more be the computational overhead of FFT. We also observed the same behavior on both of our compute platforms for running the FFT.

  All the above three factors and results we have discussed in previous chapters as well. Now we are going to discuss two new factors in this chapter. For brevity, we show results for only one of our sensing platform - RTL-SDR and one of the computing platform - Raspberry PI 1, although the similar behavior has been observed on other computing platforms as well. We use the Energy based detection techniques for the following experiments.

- **Block Size:** To see the effect of block size, we run the Energy based detection algorithm with Sample rate of 1MSps and FFT size of 512. We run our code from the minimum block size supported by RTL-SDR dongle (512) to the maximum block size (4096*16*65536). Figure 5.7 (A) shows the results for block size of 1024,2048 and 4096. We see a linear improvement in the latency after increasing the block size. The minimum legitimate block size is 1024 in this case (twice the size of FFT). A smaller block size can not be used and takes up the same latency as with the default block size(16*16*65536) which is very high (about 1250ms) for sensing time of about 10ms. For Maximum block size, latency is about 20503ms for a sensing time of 100ms.

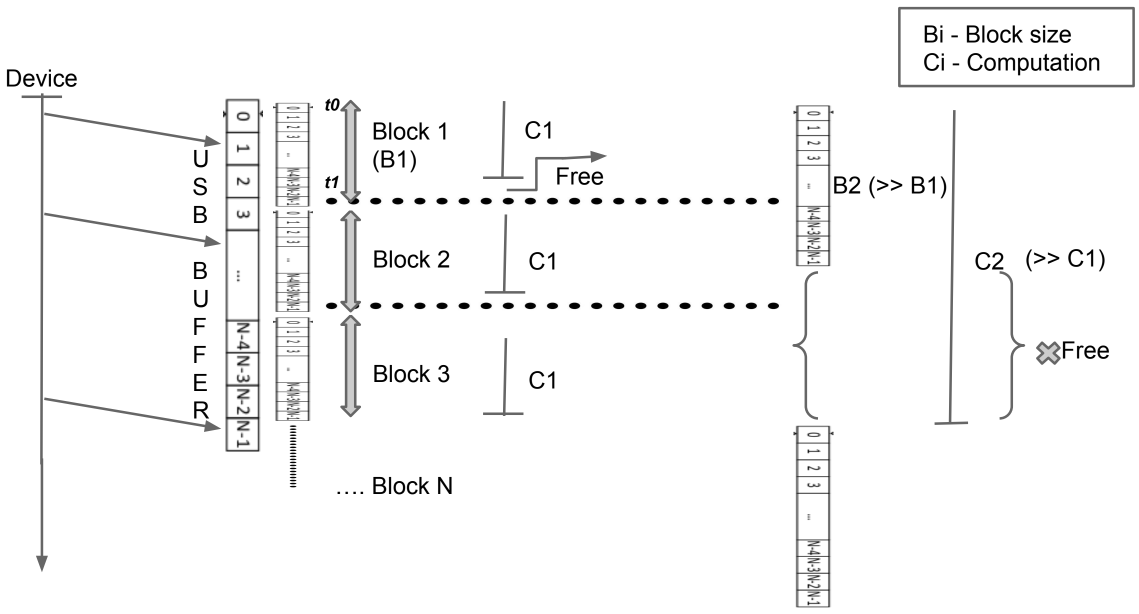
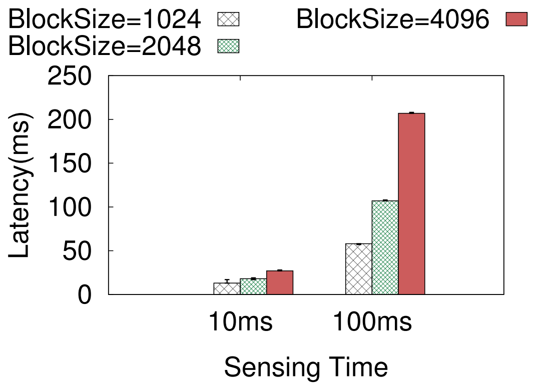


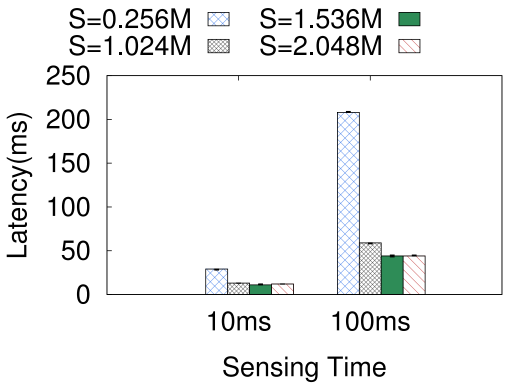
FIGURE 5.6: Higher block sizes hindering the performance of mobile sensor

In Figure 5.6, we try to explain the behavior of change in observed latency for increased block size. The USB buffer gets data from the device in terms of blocks. When block size is small, the time to perform computation on this block is also small. Note that the computation is going in parallel so if computation finishes before the next block is available, the time invested reading blocks to the USB buffer dominates in deciding total computation overhead. However when computation time is greater than the block reading time, event handler can not read next block until current block finishes the computation. This creates extra overhead for large block sizes and increases the total computational latency further.

- **Sample Rate:** We see before that for a sample rate about 1MSps, it is block size which hinders the performance but we also found that at high sample rates, it is



(A) Sample Rate=1Msps, NFFT=512



(B) Block Size=1024, NFFT=512

FIGURE 5.7: Performance Results for varying block size and sample rate

sample rate itself which affects the performance. We use the similar configurations as before for this experiment as for the block size, only difference is that we used a fixed block size of 1024 in this case and reported the results with respect to change in latency. Figure 5.7(B) shows the results for this experiment. Before we discuss the results, note that RTL-SDR does not support all sample rates. We are using Elonics E4000 tuner and only supported sample rates are 225-300 MSps and 900-2.4MSps. Elonics does not support sample rates in between 300 and 900 MSps. We did not observe any particularity in behavior till 1MSps, however after 1MSps, we see a degradation in performance. As seen in figure 5.7 (B), for a very low sample rate the latency is very high (as expected) but for 1Msps, 1.5Msps and 2.04 Msps respectively, we do not see much change in latency. We expected almost about half throughput on a double sample rate (2.04MSps) for example. We also observed that using sample rates of uneven numbers (not a power of two) have an adverse impact on the performance that may be due to architectural level optimizations for even sized blocks. We discuss and understand the behavior below.

For high sample rates, the overhead induced by sample rate itself is much higher. Figure 5.8 try to explain this behavior. Here, We look at only the timeline of one block (from $t0$ to $t1$) and see how at increased sample rate this time gets increased. In the figure, it can be seen that $(tx - t0)$ is the time taken for the computation for one block. As discussed before, if this time is less than the time to read one block into the USB buffer, the block reading dominates. However, at higher sample rate, we need to read more number of blocks in the same time that makes us do more computation in the same time (which is not possible due to CPU cycles constraints) thus increasing the overall time by a great factor. In general, we can also intuitively say from this behavior that more the sample rate, more number of blocks we need to read and more is the computation we need to perform.

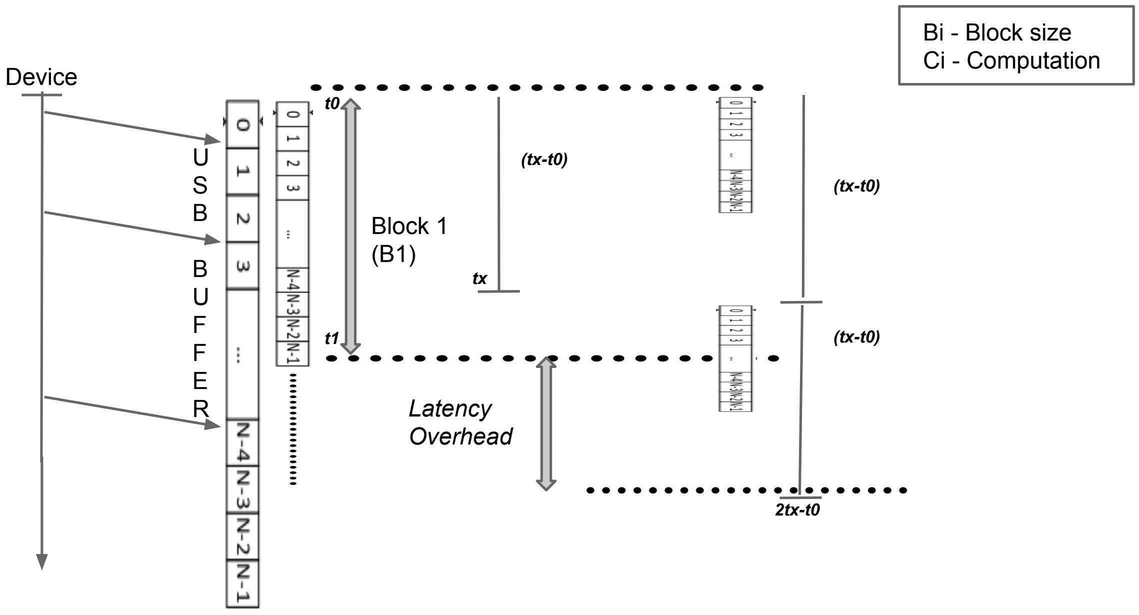


Figure 5.8: High Sample Rate hindering the computation of spectrum sensor

Chapter 6

System Design and Implementation

In this chapter, we develop a real time end-to-end indoor distributed system for mobile spectrum sensing using the prototypes discussed over the course of this thesis. We first discuss our proposed design followed by the technical challenges and design choices. We then describe the implementation details of our experimental setup and finally, wrap up the observations and experimental results.

6.1 System Design/Methodology

In this section, we describe our proposed architecture that integrates mobile sensors and Whitespace secondaries along with a cloud-based spectrum server that builds the spectrum occupancy map (or radio environment map) by collecting data from these mobile sensors and take decisions for the secondary devices to operate in the best available channel. This allows secondary devices to operate in multiple Whitespace channels depending on the availability at a particular location/time in an indoor environment.

Figure 6.1 shows a block diagram of this distributed system; we call it Wings Whitespace System. MSS1,..,4 refers to four mobile spectrum sensors. All These mobile sensors have access to the internet via any traditional (here WiFi) interface. Sensors scan a particular TV spectrum and upload the power spectral density results for individual frequency bins along with their location to the spectrum server. Spectrum server further collects this distributed data and builds a spectrum occupancy map by predicting power spectral density for other locations (points) in the area (boundaries for this area are already defined because we are in indoor environment so GPS can not be used to access the location). Now, we move a mobile secondary nearby the small Whitespace cell (within this indoor area only, See Figure) which transmits a signal in the same channel, already being used by this cell. This creates an interference between the Whitespace AP and client. The task of spectrum server is thus to detect this interference at secondaries location without actually sensing there. Based on the information obtained from other four mobile sensors in area, server can predict the power spectral density at the location of Whitespace Cell and instruct Whitespace AP to switch to another available channel (already known to server) at their location. The Whitespace AP further instructs Whitespace Client to switch to this new channel and then switches its channel also. Spectrum server communicates with Whitespace cell using any control channel (here WiFi) while Whitespace AP has two different interface - control interface(WiFi) to talk to the spectrum server and another interface capable of sensing in TV band, say Whitespace interface. Whitespace Client has only one interface; the TV band sensing interface. Whitespace AP and Whitespace Client can also use another interface to be used as control channel but we do not need it here for our purposes.

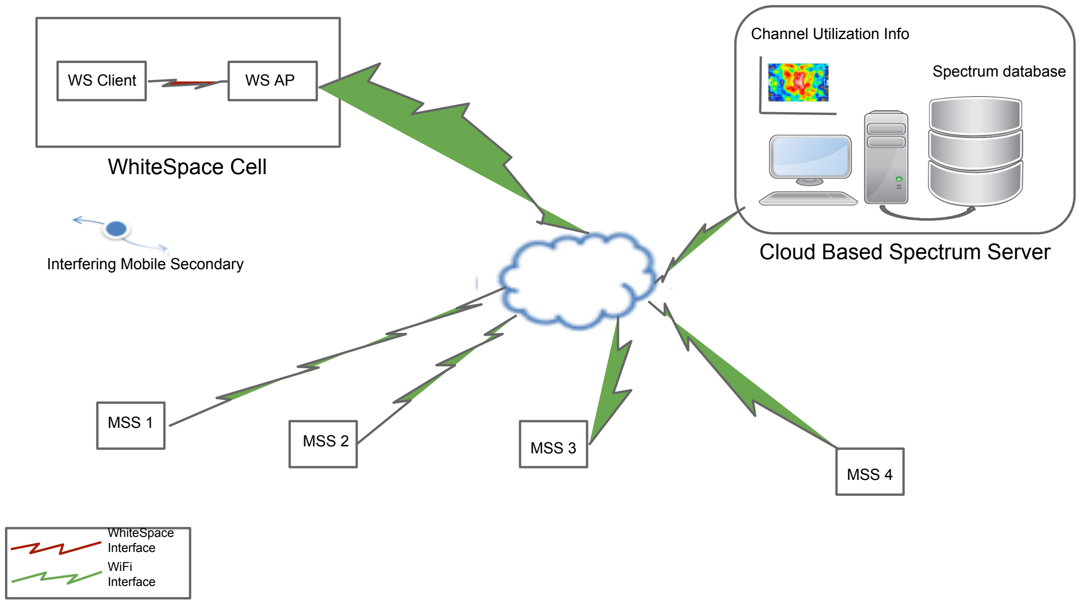


FIGURE 6.1: Wings Whitespace System Architecture

6.2 Technical Challenges/Design Choices

Before we discuss our actual experimental setups and results, we discuss in brief about different components of our testbed individually, design challenges, implementation choices and steps taken to simplify the setup.

1. The First thing to note here is that we are in an indoor environment. Spectrum Server needs to know the locations of mobile spectrum sensors to build a spectrum occupancy map, but there are no GPS like technologies available in the indoor environment. Indoor localization is a wide area of research already. As we are only developing a prototype system in an already known environment; we can create a floorplan of this indoor environment in advance. We can do so by dividing the whole area into a rectangular grid of small cells and further map this information over the two-dimensional coordinate system. Size of the cell is not of much concern, as Whitespace availability does not change much over this much short distances. Figure 6.2 shows one such floorplan used in our work.

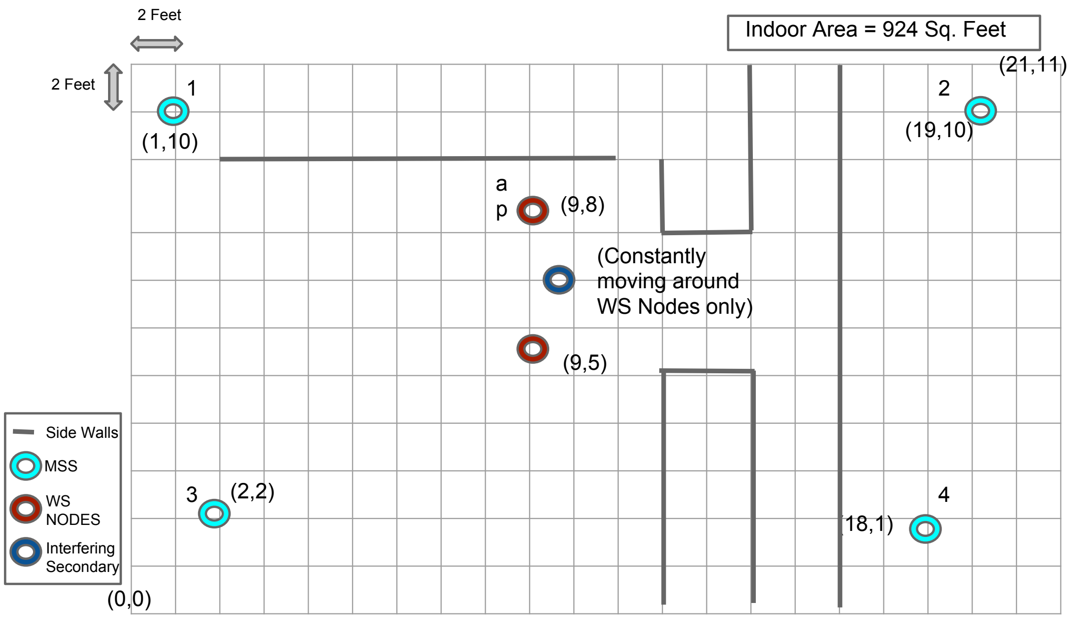


FIGURE 6.2: Floorplan

2. We mentioned before also that we are using four mobile spectrum sensors here. Although we use all four compute platforms, referred to in chapter 2 here, we limit ourselves to only one sensing device for this particular work - RTL-SDR for convenience. We discuss it later in detail.

3. Spectrum server needs to predict the power spectral density for all the locations in the indoor area by using these four values (from four mobile sensors) only. This can be done by interpolating these values over the complete 2D space, as shown in the floorplan. We used Inverse distance Weighting (IDW) [65] [66] [67] [68] for the interpolation. IDW is a well-known deterministic method for interpolation. It uses the weighted average of the known values to predict the values for unknowns. The weights are assigned as an inverse of the distance to the known point. The basic idea is fairly straightforward, more we move away from the known location, more

be the loss in signal (See path loss model [69]) and its impact on at the unknown point.

The basic equation to predict the power spectral density of all the unknown points using the IDW interpolation thus can be formed as follows :

$$P(l) = \begin{cases} \frac{\sum_{i=1}^{N} w_i(l) P_i}{\sum_{i=1}^{N} w_i(l)}, for d_{l,i} \neq 0 \end{cases}$$

where

$$w_i(l) = \frac{1}{{d_{l,i}}^p}$$

, p being a positive real number (2 in our case).

Here, $w_i(l)$ denotes the weighting function; N is the total number of points on our floorplan(12*22), l refers the unknown locations while i refers to the known locations.

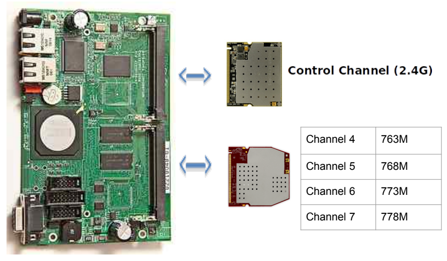
4. We have used four mobile sensors here, two being the smartphones - Nexus 5 and Samsung Galaxy S4. We have already discussed a lot about them in previous chapters. Raspberry PI being the third one is very slow in nature. The fourth device is Beaglebone Black is being used as a counterpart of Raspberry PI which may not be suitable for all applications. For all spectrum sensing devices (except RTL-SDR), we also need to deploy GNURadio driver code onto all compute platforms. Although it seems like a trivial task, it can be very challenging to install and cross-compile GNURadio for these devices due to architecture and memory limitations. There are existing images of Raspbian (The default OS for Raspberry PI) [70] and Debian/Ubuntu [71] pre-compiled with GNURadio and other necessary spectrum sensing applications. It is convenient to install these full images rather than installing the GNURadio from source by cross-compiling or from the repository. We also need to expand the file system to include the entire sdcard because GNURadio being hefty in size may need swap space to install perfectly but these devices do not have that much space on the sdcard. Default sdcard may not permit or allow this by default. We have expanded the file system to whole sdcard for our work. Although suggested sdcard size is 8GB atleast, we have used 64 GB sdcard. The more secondary storage always helps on these low-power devices. Also, it takes more than 24 hours to install the GNURadio on Raspberry PI 1 while about an hour on the BeagleBone black. This clearly shows that Raspberry PI 1 is not suitable for to operate with these high-end spectrum sensing devices(except RTL-SDR) although we have seen before that RTL-SDR works successfully and pretty well with the device. Although we were successful in running GNURadio applications using BeagleBone Black, we use only RTL-SDR

in our setup because all these sensing devices reports power spectral density in dB (not dBM) and in order to calculate the real power present at a location, we need to calibrate the device with respect to some ground truth. We preferred to avoid this for our purposes and limit ourselves to only one type of sensing device - RTL-SDR.

5. To realize the actual spectrum occupancy, server need to generate the spectrum occupancy map from the interpolated data for the whole area. This can be done by creating a Radio environment map (REM) in the form of a heatmap of the power spectral density for all locations. For generating the static heatmap, we used the R heatmap[72] library while for generating dynamic heatmap, heatmap.js [73] like libraries can be used.

6. To create a Whitespace small cell in our indoor environment, we need to have two devices that are capable of talking to each other in TV band and can be used as a Whitespace AP and Whitespace Client. there are few major concerns here :

   - The very first question is which hardware platform is better to use as the underlying hardware for AP and client. The Gateworks Avila GW234x series Processor boards [74] seems an excellent choice for developing small cell wireless networks. These are low-cost, low-power (DC 12 to 24 V, 1 A) boards that are well built and flexible as well.
   - We also need to choose an operating system to run on these boards. Although any flavor of Linux Operating System can be used ideally, using OpenWRT OS has been a preferable choice on these boards for practical purposes. The firmware program is highly optimized for size, to fit into the limited storage and memory available on these devices. The Avila boards also have small memory and storage limitations (IXP425 operating at 533MHz, 64Mb SDRAM, and 16Mb Flash memory). An OpenWRT support package is already included with these boards for Linux OS. The board also have an RS-232 Serial Port that can be utilized to get access to the OpenWRT console using a Linux Machine (with a USB port) with the help of any terminal emulation program. We used picocom [75] for our work here.
   - We also need an interface capable of scanning the TV spectrum.We have used the Ubiquiti Mini-PCI XtremeRange7 (XR7) 700MHz for our purposes. These boards provide only two Type III Mini-PCI slots, out of which one can be used for XR7 while another one can be used with any traditional WiFi Card. Although these cards do not scan in the real TV spectrum range (470-698 MHz) but it can be discounted considering the current technological limitations and making a case in the nearby spectrum range (it supports



Channel 4	763M
Channel 5	768M
Channel 6	773M
Channel 7	778M

(A) Whitespace AP using Avila Board

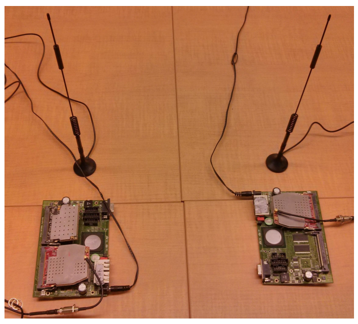
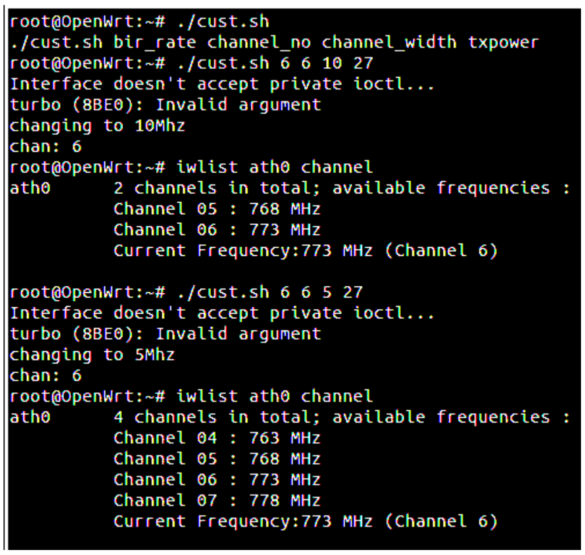

(B) Experimental Setup

FIGURE 6.3: Whitespace Access Point analogy with Avila Board and actual experimental setup

700 MHz) can also motivate the same idea. These XR7 cards support three different channel widths and four frequencies - 763M(5MHz), 768M (5,10,20 MHz), 773M (5,10,20 MHz) and 778M(5Mhz) respectively. Figure 6.3 (A) shows this anology by using an Avila board that can be used as Whitespace AP using a 2.4G WiFi card as a control channel and an XR7 card as the main channel. The board also includes two 10/100 Base-TX Ethernet ports that can also be utilized for the control interface.

- Almost all modern platforms currently rely on ath9k driver for supporting PCI/PCI-E WiFi devices. we are not using WiFi in particular which operates on 2.4G rather relying on XR7 which is not currently supported by ath9k driver. One possibility to use these cards on OpenWRT is to modify the driver code so as to support these cards but that may be an another complete piece of work in itself. Ubiquiti provides a radio driver or they call it "Ubiquiti HAL" which is a modification of the old madwifi driver. A feasible and convenient approach is to use this driver on OpenWRT OS. Everything seems great with this driver but it does not support beyond the old Linux 2.6 kernel, the current Linux kernel version supported by default in modern OSes is much higher. We need to switch back to the 2.6 kernel with OpenWRT firmware and install the modified Ubiquiti HAL to support these XR7 devices on Avila boards successfully.
- It seems like everything is setup now for our experiments but we also need to develop core applications for our work that would allow the AP and client to communicate with each other. A preferable and probably the most feasible choice is to develop C applications but as mentioned before, these boards have very less memory and storage and even standard libraries like libc are not available by default with the default OpenWRT installation. It is also not a suggested way to cross-compile these libraries for these boards. Rather,

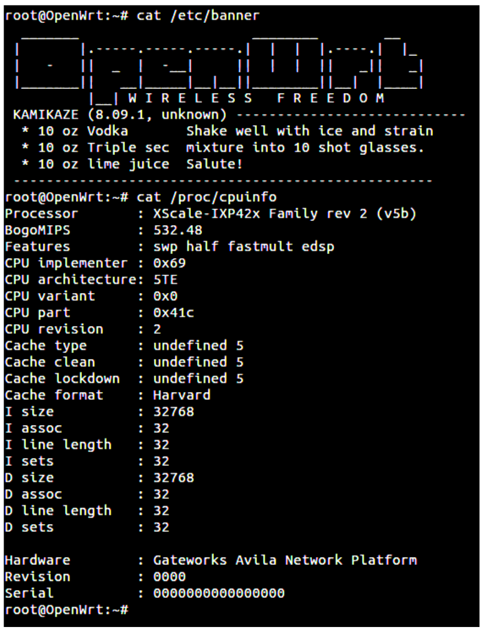


```
root@OpenWrt:~# ./cust.sh
./cust.sh bir_rate channel_no channel_width txpower
root@OpenWrt:~# ./cust.sh 6 6 10 27
Interface doesn't accept private ioctl...
turbo (8BE0): Invalid argument
changing to 10Mhz
chan: 6
root@OpenWrt:~# iwlist ath0 channel
ath0      2 channels in total; available frequencies :
          Channel 05 : 768 MHz
          Channel 06 : 773 MHz
          Current Frequency:773 MHz (Channel 6)

root@OpenWrt:~# ./cust.sh 6 6 5 27
Interface doesn't accept private ioctl...
turbo (8BE0): Invalid argument
changing to 5Mhz
chan: 6
root@OpenWrt:~# iwlist ath0 channel
ath0      4 channels in total; available frequencies :
          Channel 04 : 763 MHz
          Channel 05 : 768 MHz
          Channel 06 : 773 MHz
          Channel 07 : 778 MHz
          Current Frequency:773 MHz (Channel 6)
```

FIGURE 6.4: OpenWRT Configuration 1

OpenWRT works in a unique way here, it provides a SDK based on the architecture, OS version, processor family of the device and host architecture [1]. Figure 6.5 shows one configuration of the board when it is idle, showing the OS version(Kamikaze 8.09.1, quite old not officially supported by OpenWRT now but need to use it as we are using an older kernel) and processor information(XScale IXP42x Family rev 2, v5b).



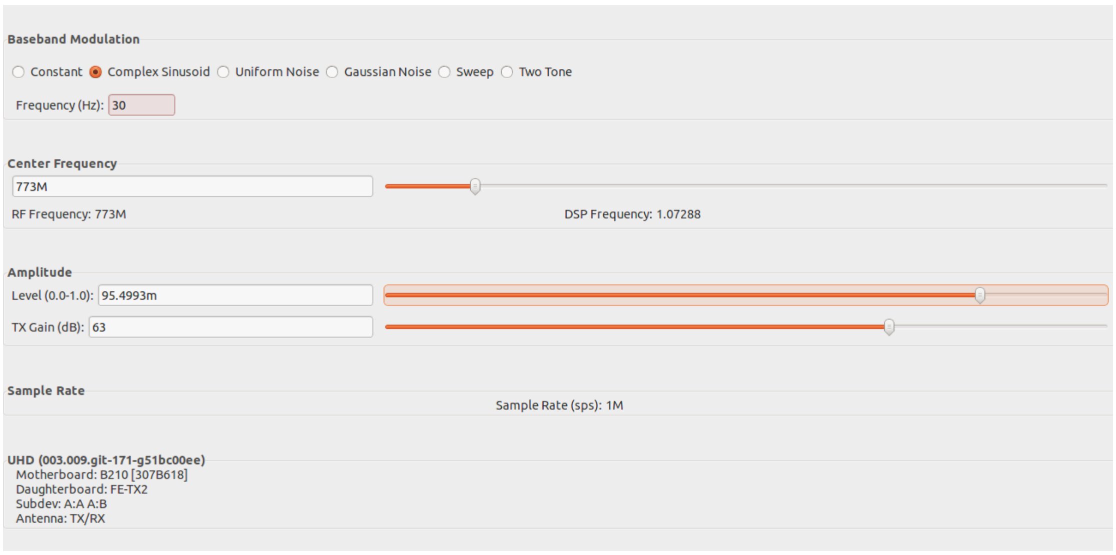
```
root@OpenWrt:~# cat /etc/banner
  _______                     ________        __
 |       |.-----.-----.-----.|  |  |  |.----.|  |_
 |   -   ||  _  |  -__|     ||  |  |  ||   _||   _|
 |_______||   __|_____|__|__||________||__|  |____|
          |__| W I R E L E S S   F R E E D O M
 KAMIKAZE (8.09.1, unknown) ----------------------------
  * 10 oz Vodka       Shake well with ice and strain
  * 10 oz Triple sec  mixture into 10 shot glasses.
  * 10 oz lime juice  Salute!
 ---------------------------------------------------
root@OpenWrt:~# cat /proc/cpuinfo
Processor       : XScale-IXP42x Family rev 2 (v5b)
BogoMIPS        : 532.48
Features        : swp half fastmult edsp
CPU implementer : 0x69
CPU architecture: 5TE
CPU variant     : 0x0
CPU part        : 0x41c
CPU revision    : 2
Cache type      : undefined 5
Cache clean     : undefined 5
Cache lockdown  : undefined 5
Cache format    : Harvard
I size          : 32768
I assoc         : 32
I line length   : 32
I sets          : 32
D size          : 32768
D assoc         : 32
D line length   : 32
D sets          : 32

Hardware        : Gateworks Avila Network Platform
Revision        : 0000
Serial          : 0000000000000000
root@OpenWrt:~#
```

FIGURE 6.5: OpenWRT Configuration 2

[1] A host machine can be any regular Linux machine where we can cross-compile and develop the applications for OpenWRT using SDKs

Based on this information, it seems straightforward to download the SDK from OpenWRT repository. There are multiple SDKs that should work based on the above information, we tried with the Kamikaze 8.09.1 (the obvious choice) SDK, Kamikaze 8.09 SDK and Kamikaze 7.09 SDK as well but we could not be able to run it successfully for cross compilation purposes. We also tried the OpenWRT-SDK-ixp4xx-2.6-for-Linux-i686 but this one could not work as well. Out of all the possibilities, only one SDK, OpenWRT-SDK-ixp4xx-2.6-for-Linux-x86_64 works finally, and we can cross-compile the applications for these boards. We can copy them using any remote application program (like scp) from the host machine to the boards utilizing the WiFi or Ethernet interface. We can install them on the boards with the help of opkg package installer (Available by default with OpenWRT). We also need to install the uclibc and libpthread for the OpenWRT using the same installer, already available in the pre-compiled SDK tools. Figure 6.4 shows another configuration of the board in which we change the channel width from the default 20 Mhz to 5 Mhz. We developed a utility called cust.sh (cust refers to the customization) which allows us to control the change the channel number, channel width, bit rate and transmit power as well.


Baseband Modulation

Constant [✓ Complex Sinusoid] Uniform Noise Gaussian Noise Sweep Two Tone

Frequency (Hz): 30

Center Frequency

773M

RF Frequency: 773M

DSP Frequency: 1.07288

Amplitude

Level (0.0-1.0): 95.4993m

TX Gain (dB): 63

Sample Rate

Sample Rate (sps): 1M

UHD (003.009.git-171-g51bc00ee)

Motherboard: B210 [307B618]
Daughterboard: FE-TX2
Subdev: A:A A:B
Antenna: TX/RX

FIGURE 6.6: Transmit a tone using GNURadio uhd_siggen_gui application for creating interference in 773M

- To create the interference, we need to transmit a signal in the same channel as used by the Whitespace cell. For this, we can use USRPs or BladeRF that can act as transmitter and receiver as well. We used USRP B210 and with the help of GNURadio uhd_siggen_gui program, figure 6.6 shows the working.

We can generate any kind of signal in general but choose to generate a tone for convenience. We can then detect the tone at the receiver end by using feature based detection that should show a high power in the transmitted center frequency. See figure 6.7 which shows the presence of a tone in 773MHz, our center frequency of the transmitted signal, received using the GNURadio osmocom_fft program with the help of RTL-SDR.

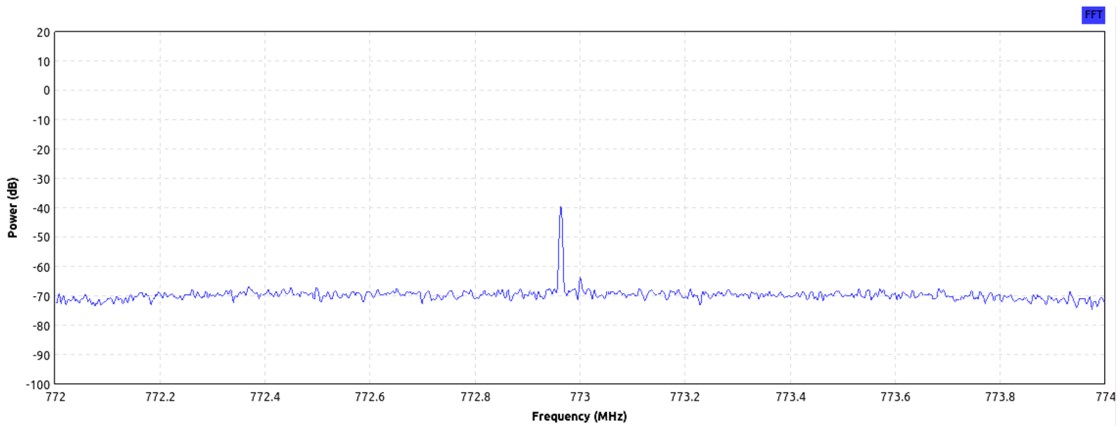


FIGURE 6.7: Receiving the signal transmitted by interfere using GNURadio osmocom_fft and RTL-SDR to detect the interference in Whitespace Cell

6.3 Experimental Setup and System Implementation

To develop the actual testbed for this experiment, we first created a Whitespace cell consists of Whitespace AP and Whitespace Client using the Avila boards running OpenWRT boards as mentioned before. Location of WS client and WS AP can be seen in the figure 6.2. The AP and client are initially talking to each other in 777M frequency (Channel 5, 5 MHz width). Now, we deployed our four sensors at respective locations (See Figure 6.2), all using RTL-SDR as the sensing platform. The device scan the spectrum keeping center frequency 773M and use feature based detection to calculate the power present in Channel 5. Spectrum Server is continuously applying interpolation over the received power values and predicts the power spectral density for Whitespace cell. Now, we start moving our interfere around the Whitespace cell white generate the interference in channel 5. The mobile sensors detect a high power with feature based detection in 777M frequency bin and report it to the spectrum server. Spectrum server also

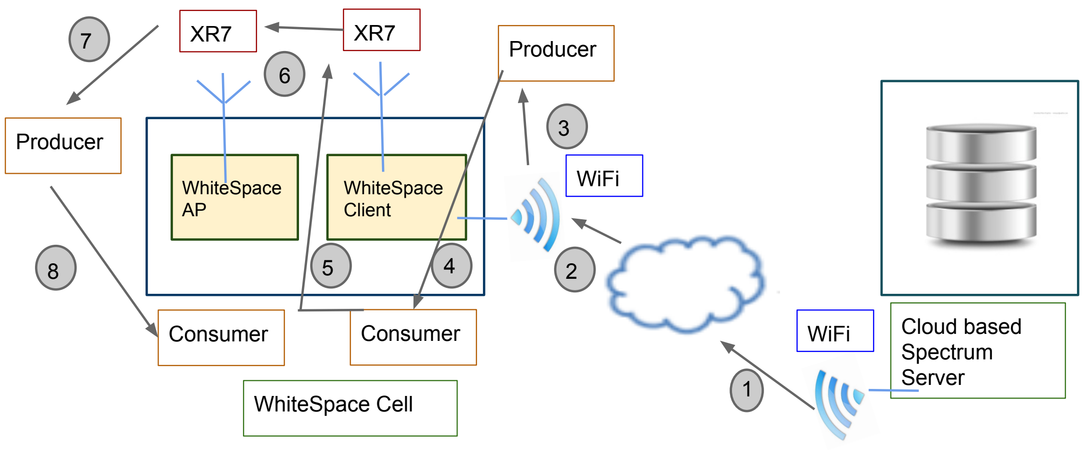
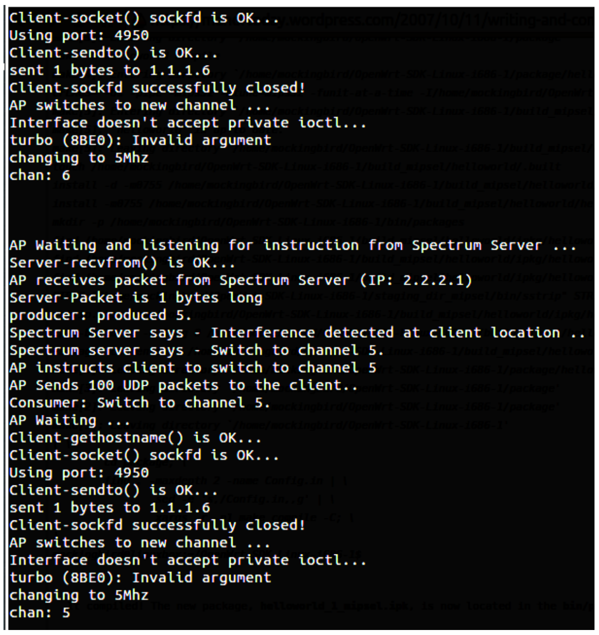


FIGURE 6.8: Communication flow between Whitespace Cell and Spectrum Server

detects (after performing interpolation) that the predicted power values are somewhat higher than the normal scenario for the Whitespace cell and makes a decision about instructing Whitespace AP to change to an available channel at the location of Whitespace Cell. In our experiments, we assumed that this new channel was already known to us (We scanned the spectrum before and verified it also). Also, Both Whitespace AP and Whitespace Client works in a producer-consumer fashion. Producer gets data from one interface and put it into the producer buffer while consumer vacates the producer buffer, use the information obtained to instruct the system for doing respective changes and talk to another interface (if required). The Spectrum server and Whitespace Cell talk over a basic UDP-based protocol to communicate further. Figure 6.8 shows the working of this protocol. The Protocol is described below.

Communication Protocol: The protocol is very simple and straightforward in nature. We have divided the complete communication into several steps. We describe each of these steps below in detail:

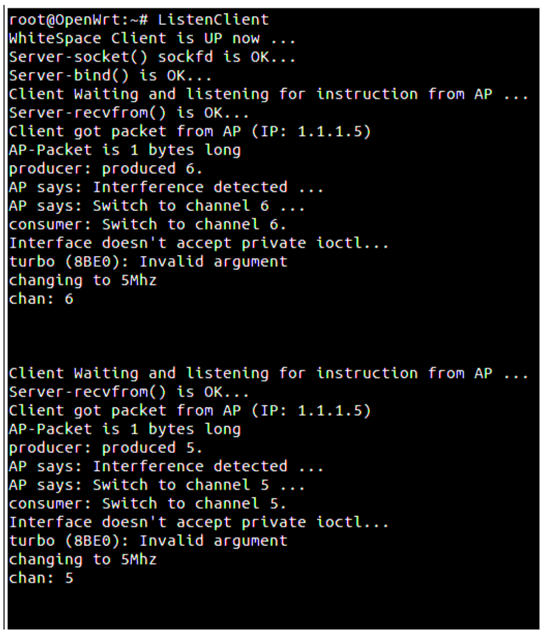
1. Spectrum server instructs and sends a UDP message to the Whitespace AP for changing to a new channel (channel 6 here) over the cloud using WiFi interface.
2. Whitespace AP receives the message over WiFi interface from the cloud.
3. Producer gets the message and put channel number into the producer buffer.
4. Consumer reads the message, vacate the producer buffer and put it into the consumer buffer.



```
Client-socket() sockfd is OK...
Using port: 4950
Client-sendto() is OK...
sent 1 bytes to 1.1.1.6
Client-sockfd successfully closed!
AP switches to new channel ...
Interface doesn't accept private ioctl...
turbo (8BE0): Invalid argument
changing to 5Mhz
chan: 6


AP Waiting and listening for instruction from Spectrum Server ...
Server-recvfrom() is OK...
AP receives packet from Spectrum Server (IP: 2.2.2.1)
Server-Packet is 1 bytes long
producer: produced 5.
Spectrum Server says - Interference detected at client location ..
Spectrum server says - Switch to channel 5.
AP instructs client to switch to channel 5
AP Sends 100 UDP packets to the client..
Consumer: Switch to channel 5.
AP Waiting ...
Client-gethostname() is OK...
Client-socket() sockfd is OK...
Using port: 4950
Client-sendto() is OK...
sent 1 bytes to 1.1.1.6
Client-sockfd successfully closed!
AP switches to new channel ...
Interface doesn't accept private ioctl...
turbo (8BE0): Invalid argument
changing to 5Mhz
chan: 5
```

Figure 6.9: Whitespace AP



```
root@OpenWrt:~# ListenClient
WhiteSpace Client is UP now ...
Server-socket() sockfd is OK...
Server-bind() is OK...
Client Waiting and listening for instruction from AP ...
Server-recvfrom() is OK...
Client got packet from AP (IP: 1.1.1.5)
AP-Packet is 1 bytes long
producer: produced 6.
AP says: Interference detected ...
AP says: Switch to channel 6 ...
consumer: Switch to channel 6.
Interface doesn't accept private ioctl...
turbo (8BE0): Invalid argument
changing to 5Mhz
chan: 6



Client Waiting and listening for instruction from AP ...
Server-recvfrom() is OK...
Client got packet from AP (IP: 1.1.1.5)
AP-Packet is 1 bytes long
producer: produced 5.
AP says: Interference detected ...
AP says: Switch to channel 5 ...
consumer: Switch to channel 5.
Interface doesn't accept private ioctl...
turbo (8BE0): Invalid argument
changing to 5Mhz
chan: 5
```

Figure 6.10: Whitespace Client

Interference	Throughput achieved in Mbps
None	1.02
Moderate Weak	0.6
Moderate Strong	0.3
Strong	0.034

TABLE 6.1: UDP Throughput Table

5. Consumer instruct the XR7 interface to switch to new channel and sends UDP message to the Whitespace Client for the same.
6. The Whitespace AP waits for some time after sensing UDP messages to the client before it changes its channel so as to ensure the delivery of message to the client and then change its channel.
7. Whitespace Client receives the message over XR7 interface.
8. Producer on Whitespace Client reads the new channel info into producer buffer.
9. Consumer vacates the producer buffer on Whitespace Client and instructs the system to change its channel.

Figure 6.9 and 6.10 shows a real communication flow between the Whitespace AP and Whitespace Client.

6.4 Observation/Experimental Results

In this section, we discuss our observations and results from the experiments mentioned above.

1. We first see the UDP throughput for the communication between the Whitespace AP and Whitespace Client in two scenarios - 1) When there is no interference and 2) When there is an interfere roaming around the Whitespace Cell. To create the interference, we further have few choices - we can transmit a very strong signal which can completely block the communication between Whitespace AP and Whitespace Client or we can transmit a very weak signal that may have no impact

between the communication. We can change the signal strength by changing the Transmitter Gain(dB) in the transmitter settings. In our experiments, we change the transmitter gain in four settings - None (Gain = 0), Moderate Weak (Gain = $\tilde{3}0$ dB), Moderate Strong (Gain = $\tilde{6}5$ dB) and Strong (Gain = $\tilde{8}0$dB). Table 6.1 show the UDP Throughput results for the four settings as indicated by running iperf UDP tests over 500 runs for each setting.

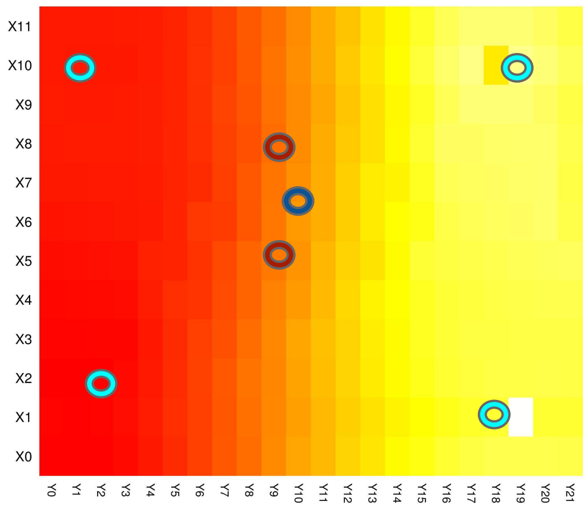


FIGURE 6.11: Heatmap

2. Spectrum Server creates a Radio Environment Map to look at the spectrum occupancy in real time by applying the interpolation over the data uploaded by four sensors. We see in figure 6.11 the individual location of all the sensors, interfere and Whitespace Cell. We see that the sidewalls present in our indoor environments have a great impact and the right side of the heatmap show almost no presence of the signal. On the left side although we see a healthy presence of signal. As mentioned before, we use the feature based detection for this experiment which we discussed in 2 also that it can detect the presence of a moderate signal. As we do transmission for a moderate strong signal for this experiment, we see that interpolation can detect the presence of signal in some amount (it is not entirely yellow but there is an effect of red also which highlights the presence of a signal). It also generate confidence in our understanding of the feature based detection that it can detect a moderate signal that may not have been detected if energy based detection would have been used otherwise.

Chapter 7

Conclusion and Future Work

The exponential increase in cellular data traffic is putting more stress day by day on the limited available spectrum. With FCC rule of deregulating the TV Spectrum for unlicensed use, there is a growing interest to exploit this spectrum in a shared opportunistic manner. The current popular approaches rely either on inaccurate model based database services or utilizing the costly and bulky spectrum analyzers that are not suitable for high granularity. We anticipate that mobile spectrum sensing may provide the tremendous amount of spectrum situation awareness with fine granularity compared to these existing solutions. However, limited computation and energy budget present on these mobile devices have always been a question of interest in the community for using these devices for spectrum sensing. In this work, we propose the design of a mobile spectrum sensor prototype utilizing modern off-the-shelf USB-powered software defined radios and commodity mobile devices. The study evaluates the performance of sensing and computation of mobile spectrum sensor. We observed different latencies and power values related to our sensing devices and numbers observed make us believe that the sensing cost is not a major concern for these devices. Retuning being one of the most important aspect consumes very little latency. Also, the power consumed by these SDRs in sensing state is also very low compared to the existing high-end devices which further motivates the idea of using these devices in a realistic scenario. As we have used all the existing low-power SDRs available in market, the comparison between various latency and power aspects among them have been provided here and can certainly help to choose the best sensing device based on the actual needs of spectrum sensing. On the other end, we also study the computation and energy overhead of well known spectrum occupancy algorithms on mobile devices and embedded computers and found them very computation intensive with respect to the sensing time in our initial serial design. Sensing time being the important accuracy aspect can not be reduce to very low, so we proposed a modified pipe-lining approach to reduce this overhead . The improved approach works great and reduce the sensing time overhead by a great factor and computation time further depends only on the FFT computation. The latency overhead for FFT execution although seems reasonable, it also starts to degrade the performance on higher FFT bins. We propose to utilize the GPU device on mobile devices for higher FFT bins and observed that GPU provides a drastic improvement for higher bin FFT. We found that energy consumed by compute devices is proportional to the latency. We also report the parameters that we believe impact the performance of mobile sensors in general and right set of value for these parameters is required for effective sensing task. We finally developed a real-time distributed system utilizing these mobile sensors where they sense and upload the spectrum occupancy information to a central cloud-based spectrum server which creates an spectrum occupancy map from this information so as to detect the interference at the location of secondaries and instructs them to switch to some other available channel at their location.

After performing this evaluation study, we believe that the entire fuzz about the computation and energy needs of mobile spectrum sensing is not a entirely true story and with right set of parameters under right constraints, mobile spectrum sensing can be used as as effective and efficient solution to realize spectrum occupancy in fine granularity for next generation networks.

In our study, we did not try to optimize and touch each and every aspect related to mobile spectrum sensing and we believe that they all can be part of the future work. In particular, a model to evaluate the performance of any software defined radio can be developed by conducting similar experiments on other devices and utilizing the information about the impact of various factors reported in this thesis. Further, The GPU used in our study is not a high-end GPU and lots of many existing mobile devices are available with so called high-end GPUs. These can be exploited further to see the improvements and applications in related areas like wideband sensing. Also the GPU implementations presented in the study have much improvement scope and may provide drastic improvements as well if address right. The system proposed in the thesis is a very basic prototype and have scopes for lot of improvements. A more sophisticated communication protocol for Whitespace communication can also be developed utilizing the information presented in this study.

Overall, we think that the biggest contribution of this work is to provide an acceptable trade-off between the accuracy and performance of spectrum sensing utilizing the mobile platforms. We hope that this study opens new set of opportunities to exploit mobile spectrum sensing for the measurement of unlicensed Whitespace spectrum.

Bibliography

[1] Cisco Visual Networking Index. Global mobile data traffic forecast update, 2014–2019 white paper, 2014.

[2] Ian F Akyildiz, Won-Yeol Lee, Mehmet C Vuran, and Shantidev Mohanty. Next generation/dynamic spectrum access/cognitive radio wireless networks: a survey. *Computer Networks*, 50(13):2127–2159, 2006.

[3] Qing Zhao and Brian M Sadler. A survey of dynamic spectrum access. *Signal Processing Magazine, IEEE*, 24(3):79–89, 2007.

[4] Fredrik Berggren, Olav Queseth, Jens Zander, Börje Asp, Christian Jönsson, Peter Stenumgaard, NZ Kviselius, B Thorngren, U Landmark, and J Wessel. Dynamic spectrum access. *Royal Institute of Technology Communication and System Department, Tech. Rep*, 2004.

[5] Ekram Hossain, Dusit Niyato, and Zhu Han. *Dynamic spectrum access and management in cognitive radio networks*. Cambridge university press, 2009.

[6] Daniel Willkomm, Sridhar Machiraju, Jean Bolot, and Adam Wolisz. Primary user behavior in cellular networks and implications for dynamic spectrum access. *Communications Magazine, IEEE*, 47(3):88–95, 2009.

[7] Rashid Abdelhaleem Saeed and Stephen J Shellhammer. *TV White Space Spectrum Technologies: Regulations, Standards, and Applications*. CRC Press, 2011.

[8] Federal Communications Commission new rulings online technical draft. https://apps.fcc.gov/edocs_public/attachmatch/FCC-12-36A1.pdf.

[9] Show my white space. Spectrum Bridge, Inc. http://spectrumbridge.com/ProductsServices/WhiteSpacesSolutions/WhiteSpaceOverview.aspx.

[10] Google Spectrum Database website. https://www.google.com/get/spectrumdatabase.

[11] Kate Harrison, Shridhar Mubaraq Mishra, and Anant Sahai. How much white-space capacity is there? In *New Frontiers in Dynamic Spectrum, 2010 IEEE Symposium on*, pages 1–10. IEEE, 2010.

[12] Ahmed Saeed, Khaled A Harras, and Moustafa Youssef. Towards a characterization of white spaces databases errors: an empirical study. In *Proceedings of the 9th ACM international workshop on Wireless network testbeds, experimental evaluation and characterization*, pages 25–32. ACM, 2014.

[13] Xuhang Ying, Jincheng Zhang, Lichao Yan, Guanglin Zhang, Minghua Chen, and Ranveer Chandra. Exploring indoor white spaces in metropolises. In *Proceedings of the 19th annual international conference on Mobile computing & networking*, pages 255–266. ACM, 2013.

[14] Tan Zhang and Suman Banerjee. Inaccurate spectrum databases?: public transit to its rescue! In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, page 6. ACM, 2013.

[15] Tan Zhang, Ning Leng, and Suman Banerjee. A vehicle-based measurement framework for enhancing whitespace spectrum databases. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 17–28. ACM, 2014.

[16] White Space Database Administrator Group. Channel Calculations for White Spaces - Guidelines (1.29). http://spectrumbridge.com/wp-content/uploads/2014/08/Database-Calculation-Consistency-Specification.pdf, 2013. Date accessed: 2014-10-17.

[17] Ayon Chakraborty and Samir R Das. Measurement-augmented spectrum databases for white space spectrum. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 67–74. ACM, 2014.

[18] Andreas Achtzehn, Janne Riihijärvi, Guilberth Martínez Vargas, Marina Petrova, and Petri Mähönen. Improving coverage prediction for primary multi-transmitter networks operating in the tv whitespaces. In *Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2012 9th Annual IEEE Communications Society Conference on*, pages 623–631. IEEE, 2012.

[19] Rohan Murty, Ranveer Chandra, Thomas Moscibroda, and Paramvir Bahl. Senseless: A database-driven white spaces network. *Mobile Computing, IEEE Transactions on*, 11(2):189–203, 2012.

[20] Anand Iyer, Krishna Chintalapudi, Vishnu Navda, Ramachandran Ramjee, Venkata N Padmanabhan, and Chandra R Murthy. Specnet: Spectrum sensing sans frontieres. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 26–26. USENIX Association, 2011.

[21] Cost of Rohde and Schwarz Spectrum Analyzer website. http://www.metrictest.com/product_info.jsp?mfgmdl=RS%20FSU50-B25-K40.

[22] Techtronix spectrum analyzer website. http://www.tek.com/spectrum-analyzer, .

[23] Metrictest - Purchase Spectrum Analyzer on sale website. http://www.metrictest.com/catalog/views/best_buy_sa.jsp, .

[24] TestEquity - Buy spectrum analyzer online website. http://www.testequity.com/categories/Network+&+Spectrum+Analyzers/Spectrum+Analyzer/, .

[25] Sayandeep Sen, Tan Zhang, Milind M Buddhikot, Suman Banerjee, Dragan Samardzija, and Susan Walker. A dual technology femto cell architecture for robust communication using whitespaces. In *Dynamic Spectrum Access Networks (DYSPAN), 2012 IEEE International Symposium on*, pages 242–253. IEEE, 2012.

[26] Ayon Chakraborty, Samir R Das, and Milind Buddhikot. Radio environment mapping with mobile devices in the tv white space. In *Proceedings of the 19th annual international conference on Mobile computing & networking*, pages 247–250. ACM, 2013.

[27] Sami Kallioinen, Mikko Vaarakangas, Paul Hui, Jani Ollikainen, Ilari Teikari, Aarno Parssinen, Vesa Turunen, Marko Kosunen, and Jussi Ryynanen. Multi-mode, multi-band spectrum sensor for cognitive radios embedded to a mobile phone. In *Cognitive Radio Oriented Wireless Networks and Communications (CROWNCOM), 2011 Sixth International ICST Conference on*, pages 236–240. IEEE, 2011.

[28] Matthias Wellens, Janne Riihijärvi, Martin Gordziel, and Petri Mähönen. Spatial statistics of spectrum usage: From measurements to spectrum models. In *Communications, 2009. ICC'09. IEEE International Conference on*, pages 1–6. IEEE, 2009.

[29] ThinkRF, the wideband Spectrum Analyzer online specs. http://thinkrf.com/wp-content/uploads/2013/05/ThinkRF-WSA4000-108-specs-130201.pdf.

[30] ESA-E Series Economy Spectrum Analyzer online specs. http://cp.literature.agilent.com/litweb/pdf/5989-9815EN.pdf.

[31] Mobile Spectrum Sensing Wings Lab Technical Draft website. https://svn.cs.stonybrook.edu/repos/whitespace/trunk/mobile_spectrum_sensing/paper.pdf.

[32] Haitham Hassanieh, Lixin Shi, Omid Abari, Ezzeldin Hamed, and Dina Katabi. Ghz-wide sensing and decoding using the sparse fourier transform. In *INFOCOM, 2014 Proceedings IEEE*, pages 2256–2264. IEEE, 2014.

[33] Haitham Hassanieh, Lixin Shi, Omid Abari, Ezzeldine Hamed, and Dina Katabi. Bigband: Ghz-wide sensing and decoding on commodity radios. 2013.

[34] Amir Ghasemi and Elvino S Sousa. Spectrum sensing in cognitive radio networks: requirements, challenges and design trade-offs. *Communications Magazine, IEEE*, 46(4):32–39, 2008.

[35] Rahul Tandra, Shridhar Mubaraq Mishra, and Anant Sahai. What is a spectrum hole and what does it take to recognize one? *Proceedings of the IEEE*, 97(5): 824–848, 2009.

[36] Sungro Yoon, Li Erran Li, Soung Chang Liew, Romit Roy Choudhury, Injong Rhee, and Kun Tan. Quicksense: Fast and energy-efficient channel sensing for dynamic spectrum access networks. In *INFOCOM, 2013 Proceedings IEEE*, pages 2247–2255. IEEE, 2013.

[37] Jincheng Zhang, Wenjie Zhang, Minghua Chen, and Zhi Wang. Winet: Indoor white space network design.

[38] Paramvir Bahl, Ranveer Chandra, Thomas Moscibroda, Rohan Murty, and Matt Welsh. White space networking with wi-fi like connectivity. *ACM SIGCOMM Computer Communication Review*, 39(4):27–38, 2009.

[39] Hariharan Rahul, Nate Kushman, Dina Katabi, Charles Sodini, and Farinaz Edalat. Learning to share: narrowband-friendly wideband networks. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 147–158. ACM, 2008.

[40] Tevfik Yucek and Hüseyin Arslan. A survey of spectrum sensing algorithms for cognitive radio applications. *Communications Surveys & Tutorials, IEEE*, 11(1): 116–130, 2009.

[41] JN Laska, WF Bradley, Thomas W Rondeau, Keith E Nolan, and B Vigoda. Compressive sensing for dynamic spectrum access networks: Techniques and tradeoffs. In *New Frontiers in Dynamic Spectrum Access Networks (DySPAN), 2011 IEEE Symposium on*, pages 156–163. IEEE, 2011.

[42] Moslem Rashidi, Kasra Haghighi, Ashkan Panahi, and Mats Viberg. A nlls based sub-nyquist rate spectrum sensing for wideband cognitive radio. In *New Frontiers in Dynamic Spectrum Access Networks (DySPAN), 2011 IEEE Symposium on*, pages 545–551. IEEE, 2011.

[43] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 280–293. ACM, 2009.

[44] Junxian Huang, Feng Qian, Alexandre Gerber, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 225–238. ACM, 2012.

[45] Ye-Sheng Kuo, Pat Pannuto, Thomas Schmid, and Prabal Dutta. Reconfiguring the software radio to improve power, price, and portability. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, pages 267–280. ACM, 2012.

[46] Prabal Dutta, Ye-Sheng Kuo, Akos Ledeczi, Thomas Schmid, and Peter Volgyesi. Putting the software radio on a low-calorie diet. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 20. ACM, 2010.

[47] Sofie Pollin, Lieven Hollevoet, Frederik Naessens, Peter Van Wesemael, Antoine Dejonghe, and Liesbet Van der Perre. Versatile sensing for mobile devices: cost, performance and hardware prototypes. In *Proceedings of the 3rd ACM workshop on Cognitive radio networks*, pages 19–24. ACM, 2011.

[48] Tan Zhang, Ashish Patro, Ning Leng, and Suman Banerjee. A wireless spectrum analyzer in your pocket. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, pages 69–74. ACM, 2015.

[49] Ana Nika, Zengbin Zhang, Xia Zhou, Ben Y Zhao, and Haitao Zheng. Towards commoditized real-time spectrum monitoring. In *Proceedings of the 1st ACM Workshop on Hot Topics in Wireless*, pages 25–30. ACM, 2014.

[50] Roberto Calvo-Palomino, Damian Pfammatter, Domenico Giustiniano, and Vincent Lenders. A low-cost sensor platform for large-scale wideband spectrum monitoring. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, pages 396–397. ACM, 2015.

[51] Shibo He, Dong-Hoon Shin, Junshan Zhang, and Jiming Chen. Toward optimal allocation of location dependent tasks in crowdsensing. In *INFOCOM, 2014 Proceedings IEEE*, pages 745–753. IEEE, 2014.

[52] D Shin, Shibo He, and Junshan Zhang. Joint sensing task and subband allocation for large-scale spectrum profiling. INFOCOM.

[53] Fast Fourier Transform. https://en.wikipedia.org/wiki/Fast_Fourier_transform, .

[54] USB Tester website. https://www.tindie.com/products/FriedCircuits/usb-tester-20-bundle/.

[55] Monsoon Power Meter website. https://www.msoon.com/LabEquipment/PowerMonitor/, .

[56] Monsoon Power Meter Manual website. http://msoon.github.io/powermonitor/PowerTool/doc/PowerMonitorManual.pdf, .

[57] FFTW Donwload website. http://www.fftw.org/, .

[58] RTL-SDR website. http://sdr.osmocom.org/trac/wiki/rtl-sdr.

[59] James A Ross, David A Richie, Song J Park, Dale R Shires, and Lori L Pollock. A case study of opencl on an android mobile gpu. 2014.

[60] Zhi Tian. Compressed wideband sensing in cooperative cognitive radio networks. In *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*, pages 1–5. IEEE, 2008.

[61] Hongjian Sun, Arumugam Nallanathan, Cheng-Xiang Wang, and Yunfei Chen. Wideband spectrum sensing for cognitive radio networks: a survey. *Wireless Communications, IEEE*, 20(2):74–81, 2013.

[62] Cooley Tukey FFT Algorithm. http://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm.

[63] Naga K Govindaraju and Dinesh Manocha. Cache-efficient numerical algorithms using graphics hardware. *Parallel Computing*, 33(10):663–684, 2007.

[64] OpenCL Fast Fourier Transform. http://www.cs.nyu.edu/courses/fall12/CSCI-GA.2945-001/dl/ruobing-report.pdf.

[65] George Y Lu and David W Wong. An adaptive inverse-distance weighting spatial interpolation technique. *Computers & Geosciences*, 34(9):1044–1055, 2008.

[66] Dale Zimmerman, Claire Pavlik, Amy Ruggles, and Marc P Armstrong. An experimental comparison of ordinary and universal kriging and inverse distance weighting. *Mathematical Geology*, 31(4):375–390, 1999.

[67] Guo-dong JIN, Yan-cong LIU, and Wen-jie NIU. Comparison between inverse distance weighting method and kriging [j]. *Journal of Changchun University of Technology*, 3, 2003.

[68] Inverse Distance Weighting wiki. https://en.wikipedia.org/wiki/Inverse_distance_weighting.

[69] Log Distance Path Loass Model wiki. https://en.wikipedia.org/wiki/Log-distance_path_loss_model.

[70] Install GNURADIO on Raspberry PI website. http://garethhayes.net/gnu-radio-for-raspberry-pi/.

[71] Install GNURADIO on Beqaglebone Black website. http://www.kd0cq.com/2014/08/packed-full-beaglebone-black-img-file-rtl-sdr-gnuradio-gqrx-lots-more-on

[72] Draw a heatmap in R website. http://flowingdata.com/2010/01/21/how-to-make-a-heatmap-a-quick-and-easy-solution/.

[73] Generate Dynamic Heatmap in Javascript website. http://www.patrick-wied.at/static/heatmapjs/.

[74] Avila GW2348 Network Processor online specs. http://www.gateworks.com/product/item/avila-gw2348-4-network-processor.

[75] Setting Up Picocom in Ubuntu website. http://processors.wiki.ti.com/index.php/Setting_up_Minicom_in_Ubuntu.