2010

Computational Complexity of Signal Processing Functions in Software Radio

Kushal Y. Shah
Cleveland State University

COMPUTATIONAL COMPLEXITY OF SIGNAL PROCESSING FUNCTIONS IN SOFTWARE RADIO

KUSHAL Y. SHAH

Bachelor of Engineering in Electronics and Communication

Gujarat University

June, 2008

submitted in partial fulfillment of requirements for the degree

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

at the

CLEVELAND STATE UNIVERSITY

December, 2010

This thesis has been approved

for the Department of ELECTRICAL AND COMPUTER ENGINEERING

and the College of Graduate Studies by

_____

Thesis Committee Chairperson, Dr. Chansu Yu

_____

Department & Date

_____

Dr. Fuqin Xiong

_____

Department & Date

_____

Dr. Wenbing Zhao

_____

Department & Date

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor Dr. Chansu Yu for giving me this wonderful opportunity and encouraging me to pursue research in the field of wireless communications. I am very grateful for his enthusiasm, patience and invaluable guidance which enabled me to fulfill my endeavors.

I would also like to thank my committee members, Dr. Wenbing Zhao and Dr. Fuqin Xiong, for being extremely supportive throughout my studies at Cleveland State University.

I would also like to thank my lab-mates, Sachin, Robert, Tianning and Murali, who provided precious inputs at various stages of my research work.

Last but not the least; I would like to thank my parents, sister, fiancée, friends, and family for their continuous support and encouragement during this entire journey.

COMPUTATIONAL COMPLEXITY OF SIGNAL PROCESSING FUNCTIONS IN SOFTWARE RADIO

KUSHAL Y. SHAH

ABSTRACT

The increased usage of mobile communication devices has imposed a challenge of achieving efficient communication with minimum power consumption. Moreover, with the advent of software defined radios (SDR), it is highly possible that signal processing functions would be implemented in software in future mobile devices. Hence, the power consumption of these future devices will be directly related to the power consumed by the processor that executes SDR software. This thesis aims at analyzing the computational complexity of different modulation schemes and signal processing communication functions of IEEE WiFi standard. This analysis provides good insight on how the computational load varies at different data rates for different modulation schemes.

For this purpose, we have analyzed computational complexity of various modulation schemes and other communication functions using widely known software radio platform i.e. USRP hardware and GNU Radio open source software platform, Matlab and OProfile (open source Linux profiling tool). After performing an extensive analysis, we are able to determine how different modulation schemes and communication functions perform computationally on a given platform. This analysis would help to achieve effective communication along with the efficient use of power in SDR based systems.

TABLE OF CONTENTS

Page

ABSTRACT…… vi

LIST OF TABLES…….. vii

LIST OF FIGURES…… viii

ACRONYMS……. ix

CHAPTER

I. INTRODUCTION…… 1

II. RELATED WORK……... 4

2.1 Low-Power Radio……... 4

2.2 Bitrate Scaling for Energy-Delay Tradeoff…… 8

2.3 Previous Work on Computational Complexity Analysis……... 12

III. SOFTWARE RADIO TESTBENCH AND OPROFILE……. 15

3.1 GNU Radio – SDR Software Architecture……. 17

3.2 USRP – SDR Hardware Architecture……. 19

3.3 Matlab and Simulink……... 21

3.4 OProfile……... 23

IV. COMPUTATIONAL COMPLEXITY ANALYSIS OF SDR…….. 25

4.1 USRP/GNU Radio-based Complexity Analysis…… 26

4.1.1 Transmitter Results with USRP1……. 27

4.1.2 Receiver Results with USRP1……. 34

4.1.3 Complexity Analysis with Bandwidth Variation……. 38

4.2 BBN 802.11b-based Complexity Analysis…… 41

4.3 Matlab-based Complexity Analysis……... 43

4.4 Summary…… 46

V. CONCLUSION AND FUTURE WORK……. 48

BIBLIOGRAPHY…….. 50

APPENDICES……... 53

A. OProfile Tools……..... 54

B. GNU Radio Signal Processing Functions…….. 56

C. Miscellaneous Commands……. 69

LIST OF TABLES

Table		Page
I.	Overall Transmitter Complexity for each Modulation Scheme……	28
II.	Summary of Modulation Specific Symbols for Transmitter…….	28
III.	Computational Complexity of Transmission functions……	29
IV.	Overall Receiver Complexity for each Modulation Scheme……	35
V.	Complexity of Individual Reception Symbols in GNU Radio……..	36
VI.	Summary of Modulation Specific Symbols for Receiver…….	37
VII.	Modulation-specific Complexity Analysis with Bandwidth Variation….	40
VIII.	Overall GNU Radio Complexity Analysis with Bandwidth Variation….	40
IX.	Profiling results of BBN 802.11b Transmitter……...	42
X.	Computational Complexity for Individual Symbols in Matlab…….	44

LIST OF FIGURES

Figure		Page
1.	Typical Hardware-Based Radio……	5
2.	Energy Delay Trade-off for QAM……	10
3.	IEEE 802.11a/b/g Implementation……..	13
4.	Block Diagram of Software Defined Radio…….	16
5.	Block Diagram of GNU Radio Architecture……	18
6.	Schematic Block Diagram of USRP…….	19
7.	Detailed Schematic Block Diagram of USRP1……	20
8.	Basic Transceiver using GNU Radio……...	26
9.	Transmitter Complexity for each Modulation Scheme……	29
10.	Computational Complexity of GNU Radio Transmitter…….	33
11.	Receiver Complexity for each Modulation Scheme……	36
12.	BBN 802.11b Transceiver……..	42
13.	Computational Complexity of BBN 802.11b Transmitter…….	42
14.	Matlab SDR…….	43
15.	Matlab SDR Transmitter…….	43
16.	Matlab SDR Receiver……..	44
17.	Computational Complexity of Matlab SDR…….	46

ACRONYMS

DBPSK	Differential Binary Phase Shift Keying
DQPSK	Differential Quaternary Phase Shift Keying
QAM	Quadrature Amplitude Modulation
CCK	Complimentary Code Keying
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
USRP	Universal Software Radio Peripheral
MAC	Media Access Control
ADC	Analog-to-Digital Converter
DAC	Digital-to-Analog Converter
CCA	Clear Channel Assessment
AGC	Automatic Gain Control
SAW	Surface Acoustic Wave
DVS	Digital Voltage Scaling
SIMD	Single Instruction, Multiple Data
PLCP	Physical Layer Convergence Procedure
SWIG	Simplified Wrapper and Interface Generator
CIC	Cascaded Integrator-Comb
GMSK	Guassian Minimum Shift Keying

CHAPTER I

INTRODUCTION

With the advent of software defined radios; it is possible to realize a fully programmable wireless communication system in the future. It is likely that in the near future, most of the mobile communication devices will be based on SDR as it can be easily reconfigured as compared to hardware radios. Most of the current SDR platforms are implemented on either Field Programmable Gate Arrays (FPGAs) or digital signal processors (DSPs). These hardware platforms are capable of supporting signal processing functions of most of the modern high speed wireless protocols. However, these hardware platforms are currently very expensive and require high skills to program them [1].

Due to above constraints, the developers often tend to use SDR systems based on general purpose processor architecture. One of the examples of such system is GNU Radio and USRP, where GNU Radio provides software platform for implementing signal processing functions on general purpose architecture and USRP provides hardware

platform which serves as RF Front end. However, currently implementing SDR on general purpose PC architecture has its own set of limitations such as requirement of very high bus throughput from RF Front end to processor, meeting low latency real time deadlines of PHY and MAC layers and able to meet high computational requirements of PHY signal processing functions [1].

As said earlier, it is expected that future mobile communication devices will be based on SDR systems. However, the computational requirements for some of the widely used wireless protocols such as IEEE 802.11a/b/g can be very high and thus can drain the power resources of the SDR devices very quickly. So besides the issue of performance, another critical requirement in future SDR devices is to manage energy usage judiciously and efficiently. In this thesis, the main focus is on the computational requirements imposed by the PHY signal processing functions on general purpose processor architectures. The aim of this analysis is to identify which signal processing functions are highly computationally intensive on the processor. This analysis would help the developers of SDR devices to select appropriate processor architecture based on the requirements of the application. Also, for a given platform, the developers can use this analysis to devise a scheme or algorithm to use the energy resources judiciously.

This thesis provides a detailed analysis of computational complexity of different modulation schemes such as M-ary DPSKs and QAMs using GNU Radio/USRP and also about the signal processing functions of IEEE 802.11b standard using Matlab/Simulink software. The rest of the thesis is organized as follows. Chapter 2 provides a background on wireless communication systems, work done on power saving mechanisms by other

researchers. Chapter 3 describes the software radio testbench used for the detailed analysis of computational complexity. Chapter 4 focuses on the computational complexity results and their detailed analysis. Chapter 5 summarizes the findings of the entire thesis. Chapter 6 discusses the future scope of this work. Usage of different software tools, details of different signal processing functions used in GNU Radio are provided in the appendix for reference.

CHAPTER II

RELATED WORK

In this section, we first discuss design considerations for low power operations in hardware-based conventional wireless radio in Section 2.1. In Section 2.2, we discuss about the impact of bit-rate scaling on energy-delay (rate) trade-off. In Section 2.3, we discuss previous research efforts on computational complexity analysis for software defined radio.

2.1 Low-Power Radio

As a reference, Fig. 1 shows a typical hardware-based radio, which consists of RF front end and electronics part [2], [3]. The RF front end is responsible for gain (power amplifier and low-noise amplifier) and frequency conversion. The electronics part is

responsible for frequency synthesis, filtering, modulation, up converting, etc. Note that, in SDR, the RF front end part still remains the same as in the conventional hardware-based radio but the electronics part is replaced by a general-purpose microprocessor-based platform with software support.

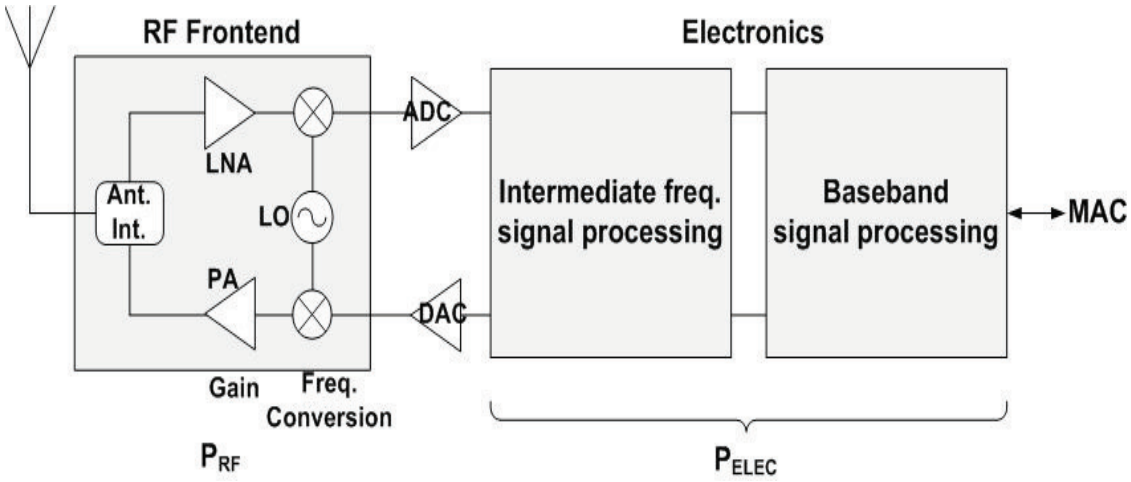

Figure 1: Typical Hardware-Based Radio

Power dissipation of RF front ends is analyzed in great detail in [4]. The main signal processing functions of an RF front end are gain (to convert the usually weak signals to convenient amplitude levels for further processing) and frequency conversion (to convert signals to convenient frequencies for further processing). In the receive path, selecting the desired channel among (many) other channels, and extracting the information that is applied through modulation to the radio signal, is usually carried out in the IF signal processing circuits. In the transmit path, modulating the information to be transmitted onto a radio signal is often also carried out in the IF circuits [4].

802.11 standards have always put power saving as priority in the design and implementations. Initial implementations of the 802.11 standard including Prism I and II attempt to reduce the energy cost in many different ways. One of the ways to achieve power savings is to shut down the radio when it is not being used. The media access controller (MAC) keeps sensing the channel for any signal and if there is not activity then it turns off the radio. It is also possible to put different parts of communication circuitry to sleep using separate power control lines in order to save power. It is also possible to save power by transmitting at as low power as possible without compromising the reliability factor. This transit power control has something to do with modulation scaling as discussed later in this section.

Design considerations for low power WLAN in the framework of 802.11 standards have been discussed in detail in [5]. Minimizing power consumption is one of the important features of IEEE 802.11 standards. IEEE 802.11b standard is designed for transmitting at lower distances at higher data rates such as 1, 2, 5.5 and 11 Mb/s. Processing gain and multipath protection are achieved using efficient phase shift keying (PSK) waveforms. By minimizing the time frame when transmitter is on and transmitting with minimal power usage would be the key objective of power reduction.

Some of the well-known methods for saving power in 802.11 standard are to transmit at as low power as possible, operating at low voltage, sensing channel at low power for low power acquisition, putting radio to sleep when unused, and using single oscillator and surface acoustic wave (SAW) filter [5].

Different sections of the circuitry can be put to sleep by using separate power control enable lines. These sections are generally the ones used in conventional radio such as RF and IF processing sections. Even the gaining circuitry is kept running at low power until significant activity is sensed on channel in order to contribute to power saving. In order to achieve this signal detection is carried out by detectors in baseband processing section. One of the major power consumption areas in a communication system is sensing the channel and processing it continuously when radio is in receiver mode. So it is very important to minimize power consumption as much as possible in this section. Two techniques used for carrier sense-processing are Clear-channel-assessment (CCA) and acquisition activation. Automatic Gain Control (AGC) behavior and Barker codeword correlation can be used together in order to sense carrier on channel. Thus AGC can be used in carrier sense process to minimize power consumption to a large extent. Deciding a threshold level for starting the digital signal processing would offer significant advantage to power saving. For example, if in a certain application very weak signals are not important and can be ignored without any impact on performance, then a huge amount of power savings can be realized as the receiver circuitry would be activated only when there is sudden rise in received signal power probably indicating arrival of a packet. However, if the weak signals are also important for the performance then Barker correlation should be used for carrier detection. For example, a signal transmitted at a data rate of 1 Mb/s can have an SNR of 0 dB and hence in this case Barker correlation on the noise floor becomes very important. Barker correlation requires only additions and subtractions and hence can be easily implemented such that it consumes low power [5].

802.11 standards also have low-power acquisition mode and Power Save Mechanism (PSM) which can address energy cost during the long, idle listen period. The radio can end up spending more energy while listening idly for a signal. This cost can be minimized by turning on only the low-power carrier-sense processing while not running the costly acquisition circuitry [5].

Still, idle listening could be a significant contributor of the total energy expenditure simply because of its longer duration. Low power sleep state is available in many conventional radios. In this state, significantly low power is consumed as even carrier sense processing is not allowed [6]. Low power sleep state has shown clear benefits over various different energy cost studies. IEEE 802.11 PSM allows a radio can go into sleep mode on its own if it has nothing for transmission or reception.

2.2 Bitrate Scaling for Energy-Delay Tradeoff

The major contributor of power consumption in software radio is the radio electronics used for transmissions at GHz carrier frequency, which is particularly true for short-range communication (RF Front-end power, P_{RF}, on the order of 1mW for signals transmitted at 1Mbps with BER of 10^{-5}) [3]. Sending frames in burst at high bit rates and then turning off transmitter during no activity periods can also help in power savings. However, this method causes a significant overhead for switching between on and off state [3] and thus, a better alternative is to use dynamic voltage scaling (DVS).

Each transmitted symbol in M-ary modulation scheme is obtained from a number of distinct waveforms. Thus, $\log_2 M$ number of bits are required for each symbol. Some of the well-known M-ary modulation schemes are Phase Shift Keying (M-PSK), Quadrature Amplitude Modulation (M-QAM), and Frequency Shift Keying (M-FSK). In [3], bandwidth efficiency is analyzed against power used for transmission. In order to carry out this analysis they transmitted a signal at 1Mbps with a frequency of 5.8GHz and BER was set to 10^{-5}. The channel was assumed as Rayleigh fading channel. The path loss exponent is around 2-3 as can be seen in a closed environment with large obstacles [3]. In M-ary modulation schemes, transmit on-time is minimized by the number of bits per symbol if the symbol rate is kept at 1 MSymbols/s. However, this can increase both P_{RF} as well as P_{Elec} [3].

In [7], modulation scaling and the corresponding energy-delay tradeoff has been studied in detail

$$E_{bit} = (P_{RF} + P_{Elec}) * \tau \quad (1)$$

Where τ is $\frac{1}{R_s * b}$, R_S is symbol rate, and b is the constellation size (number of bits per symbol).

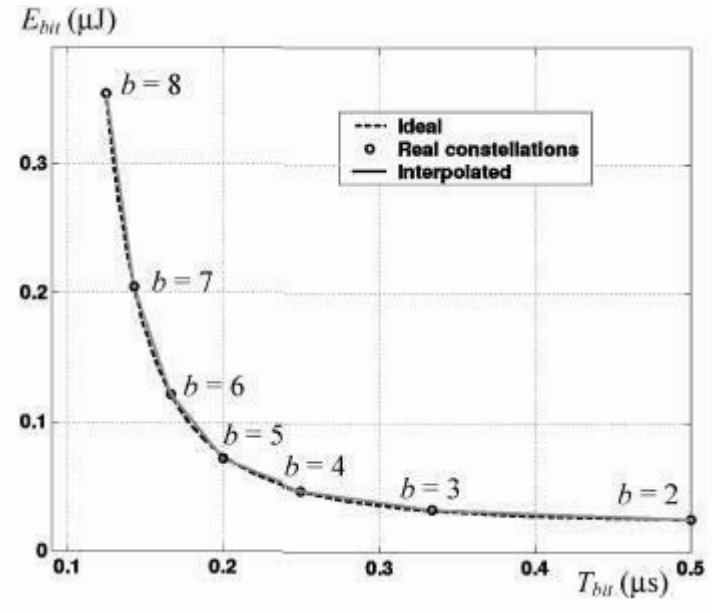

Figure 2: Energy Delay Trade-off for QAM [9]

As b increases, the packet delay (τ) is reduced but energy cost (E_{bit}) increases. In this case certain portions of the circuitry runs at a frequency which is close to instantaneous rate of symbols (R_S) while the other portions run at maximum possible symbol rate, hence the power consumed by the electronic circuitry can be represented as follows [7],

$$P_{Elec} = C_E R_S \quad (2)$$

P_{RF} depends on the BER requirement related to the modulation scheme used. The value of C_E is affected by radio architecture, circuit implementation and the semiconductor technology. In other words,

$$P_{RF} = C_S R_S (2^b - 1) \quad (3)$$

Fig. 2 shows the similar trend [8]. Here, C_S is also constant based on receiver implementation and operating temperature. It is clear from above discussion that by varying constellation size 'b' one can find an optimal point between energy consumption and performance delay. This process is known as modulation scaling. The constellation size can be modified in order to minimize power consumption such that it does not increases delay to an extent that it affects the performance. In [7], this tradeoff has been exploited to develop an energy-aware packet scheduling scheme. It is possible to conserve energy in packet scheduling system by bringing modulation level down when there are no packets in queue. And similarly when more number of packets queue up, one can increase 'b' so that there is no overflowing or long queuing. The basic concept is to adjust the 'b' of the modulation schemes based on the traffic in the system (i.e. being transmitted).

The energy-delay tradeoff has been discussed in [7]. Rate or constellation size adaptation helps to improve the energy and delay performance. It also proposes a link scheduling algorithm in the context of TDMA-based sensor networks [9]. They are based on the prior result that for some short-range applications, M-ary modulation outperforms binary modulation for energy savings by decreasing the transmission time, which again assumes that transmit power is adjusted according to the constellation size to maintain BER.

In modern wireless nodes, three critical parameters are present all the time, which are as follow: voltage scaling, convolution code strength, and radio transmission power [10]. For example, based on the required transmission distance one can adjust the transmitter power. Also, this study uses four criteria to judge the efficiency of communication

system: range, reliability, latency, and energy. These are the most general criteria used to specify communication requirements of an application.

2.3 Previous Work on Computational Complexity Analysis

The most recent study on computational complexity has been conducted in [1], which was necessary to assess the performance limitation and to comprehend abundant new ideas such as lookup tables (LUT), SIMD (single instruction, multiple data) architecture, etc. SORA is a software radio platform that can realize the commercial 802.11a/b/g network interface cards in combination with SoftWiFi radio system [1]. It includes radio control board (RCB) which connects PC memory through high-speed and low-latency PCIe bus to RF front-end, SIMD extensions in existing processors, software architecture that uses lookup tables aggressively, and real-time provisions for faster PHY processing.

According to their performance study, receiving signals modulated higher modulation rates requires higher CPU utilization. It is observed that a single core of present day's multi-core CPUs can easily handle load imposed by different 802.11b modulation modes. Sora SoftWiFi requires about 70% of the total power available from a single core for processing SDR functions at a data rate of 11Mbps. However, two cores might be used for processing receiver functionalities of 802.11a/g PHY layer. From their study, Viterbi decoder proves to be the most computationally intensive section of 802.11a/g statndard. It requires about 1.4 Gcycles/s when modulation data rata is higher than 24Mbps [1].

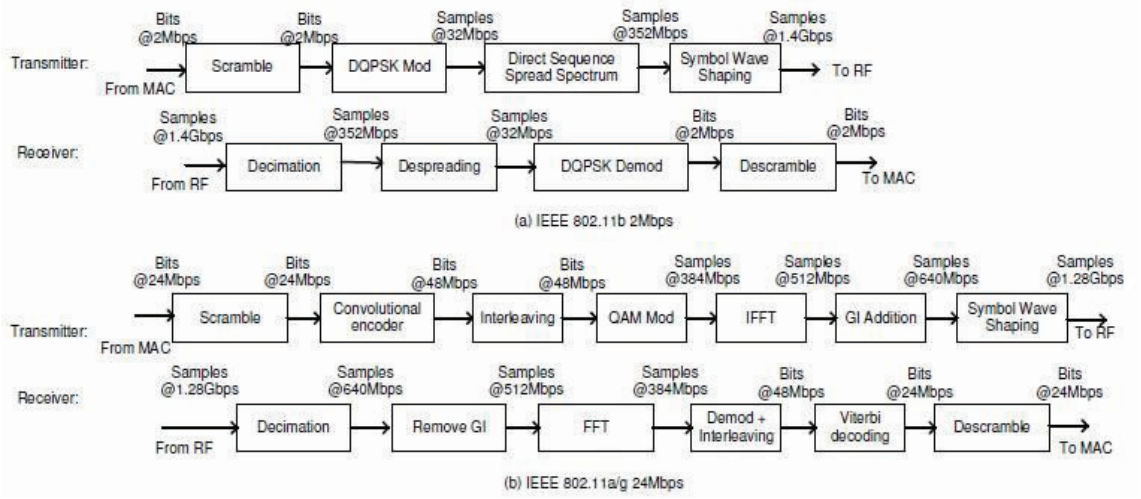

Figure 3: IEEE 802.11a/b/g Implementation [1]

The least computationally intensive component found in 11b and 11a/g standard was the Frame detection section. It required only 11% for 11b and 3.2% for 11a/g and it remains constant for different data rates. However, one important thing to note is that the frame detection takes place every single time even if there is not useful communication as the receiver does not know when the frame will arrive. Also in case of SORA, whenever a frame is detected it utilizes about 29% of a single core to synchronize in 11b and about 20% of a single core in 11a/g [1]. The next step that SORA performs after frame synchronization is it will demodulate Physical Layer Convergence Procedure (PLCP) header. This header is always transmitted at the lowest data rate supported by the 802.11 standard. This component requires about 27.5% of a single core in case of 11b and requires about 44% in case of 11 a. Thus from their study, it is very clear that demodulation at higher data rates is the most computationally intensive part of a communication system. Also, theoretically they found that direct implementation of 802.11b requires about 10 Gops while 802.11a/g requires about 40 Gops. Software techniques in SORA are efficient PHY processing by pre-calculating LUTs (make them

resident in L2 cache), multi-core streamlines processing, and real-time support. Also, they found that SIMD model can easily accommodate FFT, FIR filters and Viterbi decoder. FIR filter is the most demanding in the implementation of 802.11b while Viterbi decoder is the most demanding in 802.11a.

To SDR platform developers, it is important to understand the computational complexity of SDR functions in order to make a decision on architectural choices for the SDR. For example, SODA is software radio platform, based on an asymmetric processor consisting of a scalar and SIMD pipeline [11]. According to [11], the heaviest computational work is Viterbi decoder, FFT and IFFT in 802.11a (24Mhz) wireless system. In W-CDMA, it is Searcher and Turbo decoder. In [12], computational complexity of DQPSK modulation has been analyzed in the context of GNU radio/USRP. Filtering in receiver is highly computationally intensive, ranging from 100-200 operation/sample. Applying the sampling speed of 22.5 MHz will result in 22.5 MSPS (mega samples per second), so that it needs 2,250 - 4,500 MIPS (million instructions per second).

This thesis, for the first time, evaluates the computational complexity of SDR software by using the modulation schemes implemented in GNU Radio, BBN implementation of 802.11b in GNU Radio and 802.11b implementation in Matlab. This thesis evaluates the communication complexity of SDR in terms of the total cycles as well as the instantaneous CPU power (cycles per second) to correctly compare the complexity of different modulation schemes. For the microprocessor that runs SDR, the latter is more important because it determines the voltage and frequency level and thus, the energy cost.

CHAPTER III

SOFTWARE RADIO TESTBENCH AND OPROFILE

In the recent times, software defined radio has gained a lot of importance as it provides flexibility to the radio communication by implementing radio functionality in software rather than in hardware. Some of the major advantages of software radio are that they can be reconfigured “on-the-fly”, their features can be quickly and easily upgraded, and they can be used to build smart or cognitive radios. However, due to the constraints of today’s technology, there is still some RF hardware involved in software defined radio system. Figure 4 illustrates the block diagram of software defined radio. There are quite a few software defined radio systems today and one of them is GNU Radio/USRP system. The emergence of GNU Radio software and USRP hardware has allowed the

research community to develop and analyze wireless communication systems easily in software radio environment.

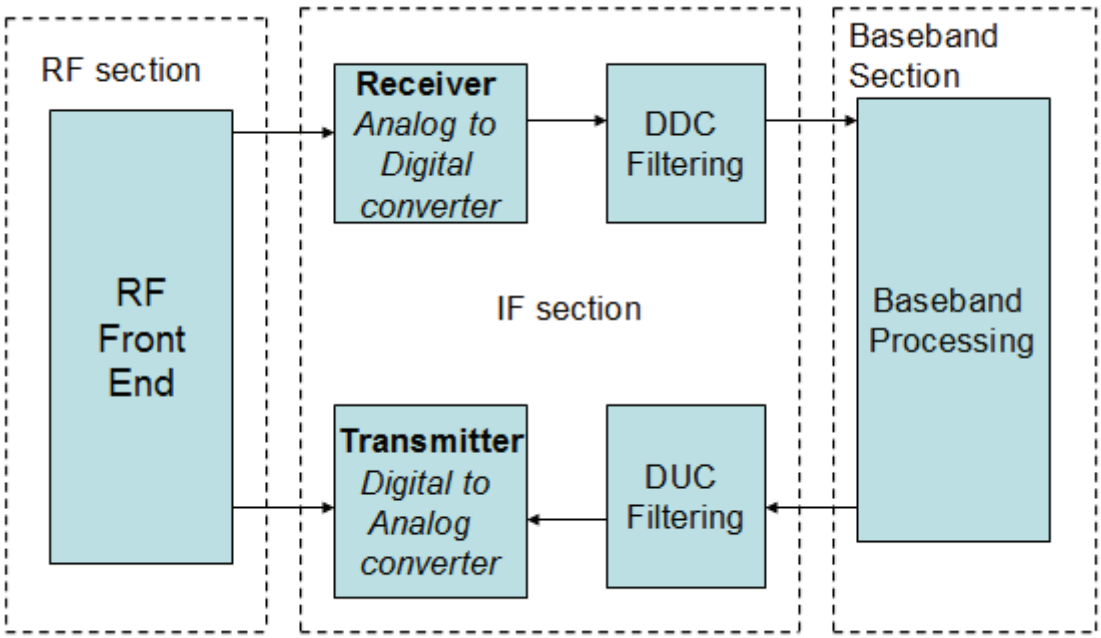

Figure 4: Block Diagram of Software Defined Radio

GNU Radio is an open source software toolkit that allows easy development of software defined radios. GNU Radio provides signal processing blocks which can be used to implement software radio functionalities on a general purpose processor. Radio front-end for GNU Radio is provided by USRP. USRP acts as a flexible hardware platform that can provide basic RF front-end functionalities.

3.1 GNU Radio – SDR Software Architecture

GNU Radio software architecture provides a library of signal processing blocks which can be glued together to build and deploy software defined radios [13]. The library provides various functions for signal processing functions such as filtering, adding signals, transforming, decoding, hardware access and many others. These libraries or modules are implemented in C++ language in the form of classes. Each of these signal processing block is equivalent to complex communication block implemented in conventional hardware radio. Thus, in GNU Radio, low level communication blocks are implemented by these C++ modules. In GNU Radio, the top level application programming for implementing advanced wireless radio communication protocols is implemented in Python scripting language. Basically, Python is used to create a flowgraph to connect signal processing blocks in GNU Radio. This flowgraph resembles to a radio chain comprising of nodes which are the signal processing blocks implemented in C++ while the data flows along the edges of the flowgraph. Each node in the flowgraph performs exactly one signal processing function while the data flowing along the edges of the flowgraph can be in the form of symbols, samples or bits. Each flowgraph at least needs one source and one sink acting as input and output ports. Figure 5 provides the generic block diagram of GNU Radio Architecture.

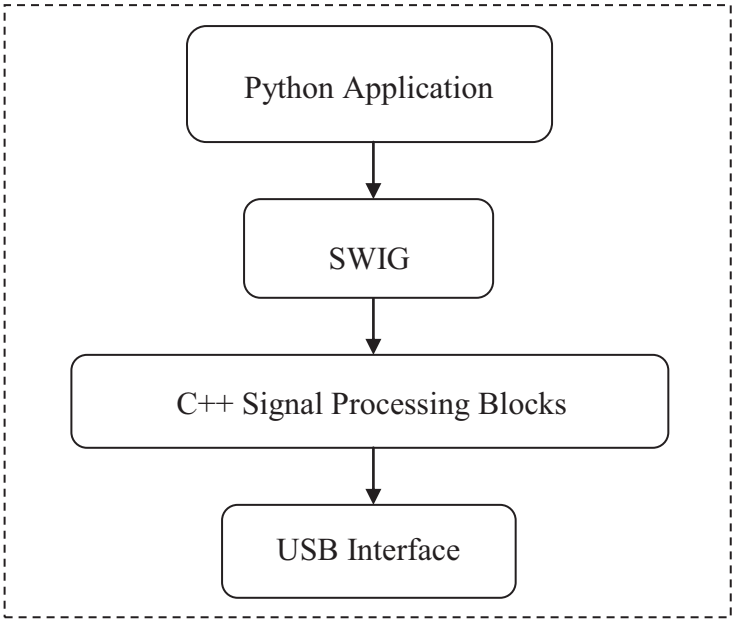

Figure 5: Block Diagram of GNU Radio Architecture

As the signal processing blocks and higher level application development is done in two different languages, SWIG (simplified wrapper and interface generator) is used to connect them together. SWIG is a software tool which wraps C/C++ blocks for using them with a variety of high level scripting languages such as Python, TCL, Perl and many more. Due to the use of SWIG, GNU Radio architecture is capable of using powerful features of both Python and C++ languages [14].

C++ is used to implement signal processing blocks it can efficiently manipulate bytes, packet headers, and implement algorithms that can run over large data sets. On the other side, Python is used for its flexibility and ease of programming as it allows developers to build their applications quickly. Together Python and C++ blocks can implement software radio functionalities on a general purpose processor. USRP hardware provides or accepts data from GNU Radio through USB cable connected to computer.

3.2 USRP – SDR Hardware Architecture

While GNU Radio provides the software platform for implementing most of the signal processing functions, Universal Software Radio Peripheral (USRP) provides a basic hardware platform in order to transmit and receive signals at different frequencies with different bandwidths. In short, USRP provides RF frontend for software defined radio platform. USRP consists of a motherboard which can support different daughterboards for communication at different frequencies. It was developed by a team headed by Matt Ettus [15]. There are two different versions of USRPs called as USRP1 and USRP2. USRP1 consists of RF frontend, ADCs/DACs, FPGA and USB controller while USRP2 is a more advanced version of USRP1 and it provides Ethernet connectivity to the computer instead of USB connection. Following figure shows schematic block diagram of USRP1.

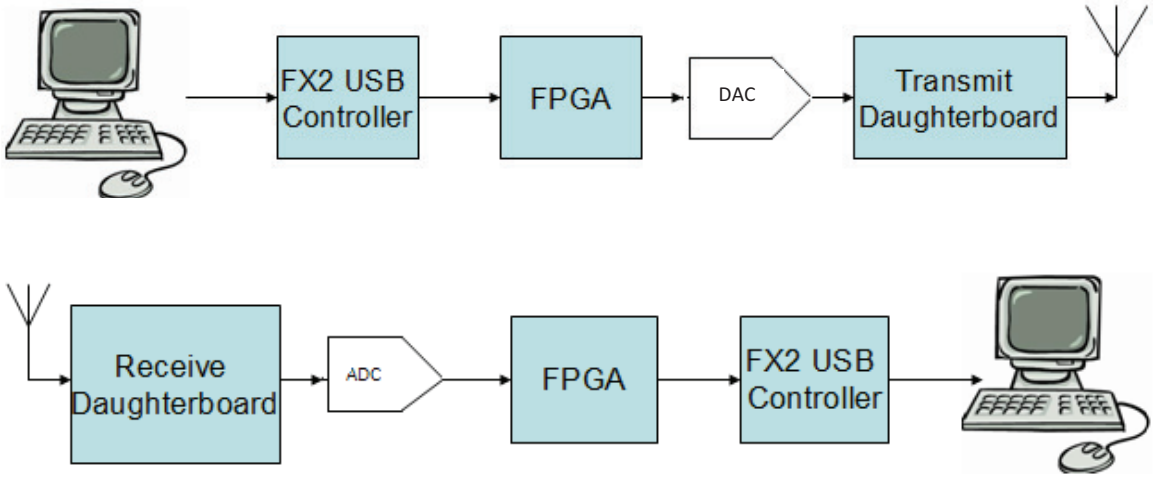

Figure 6: Schematic Block Diagram of USRP

As seen in the figure above, USRP1 uses a FPGA for performing frequency up/down conversion. Basically, FPGA manages the data rate of the signal so that it can be

transferred through USB cable to or from the computer. ADCs/DACs are respectively used to convert signal from analog to digital format and vice versa. USB controller controls data transfer over USB cable [16]. All of the above things are accommodated on the motherboard while the daughterboards provide the RF front-end functionality. Depending on the frequency band to be used for communication, different daughterboards can be plugged in and out of USRP system. The figure below shows a detailed schematic block diagram of USRP1.

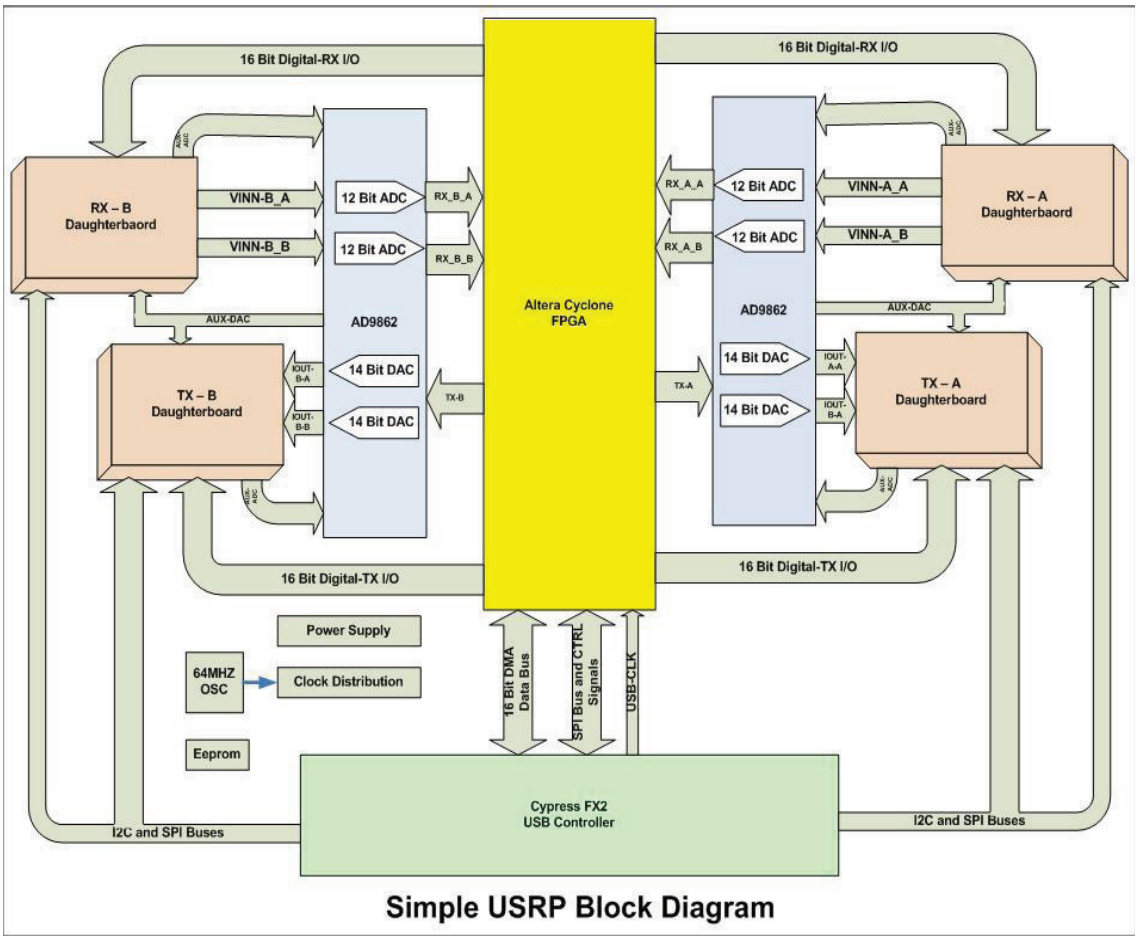

Figure 7: Detailed Schematic Block Diagram of USRP1 [17]

In case of USRP1, the digital down converters used in RX path are implemented in FPGA configuration while the digital up converters used in TX path are implemented in AD9862 CODEC chips instead of FPGA. And the only transmit signal processing blocks implemented in FPGA are the CIC (cascaded integrator-comb) interpolators [17].

Thus, in short, both GNU Radio and USRP were used together to build software defined radio in this thesis and were used to analyze computational complexities of different signal processing functions implemented in software. In the next section, Matlab and Simulink software is covered in detail as they are very useful and effective tools for simulating software radio functionality.

3.3 Matlab and Simulink

Matlab is a very high level language that provides an interactive environment so that one can easily focus on their applications rather than worrying about the programming details. Because of its flexibility and wide reach, Matlab is used to implement numerous applications in field of engineering, science and mathematics. Some of these widely known applications are signal and image processing, communications, control design, financial modeling and analysis, and computational biology. Moreover, Matlab provides some toolboxes which are collections of task and application specific Matlab functions, which makes application development a lot easier for the developer.

It also makes it easier to develop various algorithms and architecture exploration for communication systems. It allows design teams working in different areas such as RF,

baseband, control, and analog to collaborate easily [18]. Also, through re-use of models and algorithms already available in Matlab, it is possible to enable early verification throughout the design cycle. It also allows integration with legacy code and with third-party hardware and software co-simulation environments easily. For above mentioned reasons, Matlab is a great tool for designing and simulating communication systems.

On the other hand, Simulink can be used for modeling, simulating, and analyzing dynamic systems in multiple domains that include controls, signal processing, communications, and other complex systems [18]. Moreover, modeling in simulink is easy as it provides graphical user interface and a customizable set of block libraries.

For computational complexity analysis, Simulink is used to implement software defined radio applications as it allows simulation and performance analysis of SDRs. Simulink also provides automatic code generation for creating embedded software. Also, tools provided in simulink can easily interact with SCA and VHDL code generators. Simulink can also be used for specification capturing and executable implementation-independent model construction. Moreover, it allows model elaboration from behavioral modeling using fixed-point analysis [18].

Simulink enables designer to build implementation-independent models required by SDR programs and hence allows code portability and reuse. Moreover, there are some useful toolboxes that are readily available along with Matlab and Simulink. The most important toolboxes that were used in this thesis for software defined radio purposes were communications toolbox, signal processing toolbox.

Signal Processing Toolbox and Blockset provide tools for design and analysis of industry-standard algorithms for analog and digital signal processing. Communications Toolbox and Blockset provide tools for exploring, analyzing, designing and simulating physical layer of communication systems [18].

3.4 OProfile

In order to analyze computational complexity of communication functions, it was necessary to use a system profiler that collects the information with very low overhead. OProfile seemed to be the best choice as it has been used in many previous research works and it is widely used for Linux systems with a wide variety of underlying processor architectures.

OProfile uses hardware performance counters already available for various events in the processor for profiling application code. It supports all the Intel processors (32-bit as well as 64-bit), AMD Athlon, AMD64, ARM, Alpha and more. It also works on most 2.2, 2.4 and 2.6 kernels.

OProfile works on the principle of sampling, and it helps the programmer to identify problems with their code [19]. OProfile uses a kernel driver which is well supported by daemon to collect data which is profiled by it [20]. It also provides several tools which can be used for interpreting the raw data collected by the profiler. OProfile makes use of the hardware performance counters available in the CPU. It can also be used to measure time spent by each function – a functionality provided by gprof [20]. The tools available

in OProfile for post profile analysis that allows user to generate function-level or instruction-level detailed reports. OProfile imposes very low overhead on the system which is very beneficial. This overhead varies based on the sampling frequency.

Some of the common events that OProfile can monitor are total number of retired instructions, time during which processor is not halted, retired branches, retired mispredicted branches, cache references, etc.

CHAPTER IV

COMPUTATIONAL COMPLEXITY ANALYSIS OF SDR

The purpose of this section is to obtain the computational complexity of SDR functions in terms of the number of samples (instructions executed by processor). However, for the microprocessor that runs SDR, what is more important is the required instructions to execute per second because it determines the voltage and frequency level. In section 4.1, 4.2, and 4.3, computational complexity of USRP/GNU Radio, BBN 802.11b and Matlab is analyzed respectively. In section 4.4, we provide a brief summary of our observations. It is noted that our evaluation results using USRP, GNU Radio and Matlab coincide with observations made by other researchers mentioned above in [1], [11].

4.1 USRP/GNU Radio-based Complexity Analysis

The following are the details of the experimental setup. (i) Each USRP system (version 5b) includes a RFX2400 transceiver (2.3-2.9 GHz) and GNU Radio software (version 3.1.3). (ii) Modulation schemes profiled on the transmitter side are GMSK, DBPSK, DQPSK, and QAM while only GMSK, DBPSK, and DQPSK are profiled on receiver side. (iii) Carrier frequency and bandwidth we have tested are 2.4 GHz and 100 KHz, respectively, with the data rate of 100 - 1000 Kbps. A smaller bandwidth and data rates are used partly due to bandwidth constraints imposed by the USRP [21]. (iv) Transmitter amplitude is set to 8,000, which is smaller than the default value (12,000). (v) Packet size is 1,500 bytes and 1,000 packets were transmitted for each experiment. (vi) We used Oprofile as a profiling tool. (vii)We profiled transmitter as well as receiver complexity using benchmark_tx.py in GNU Radio. See Fig. 8 for the corresponding signal flow.

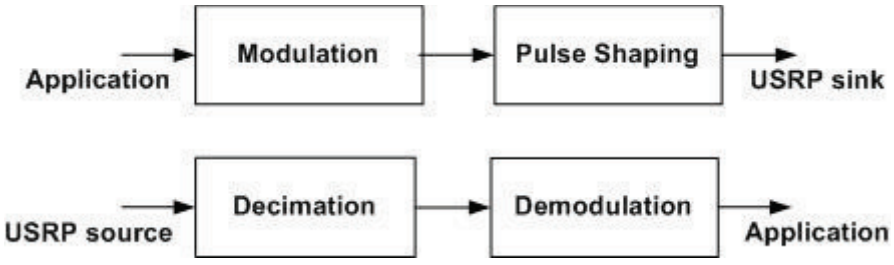

Figure 8: Basic Transceiver using GNU Radio

We describe only parts of the whole symbols because the number of the profiled results is too many. These symbol names allow us to understand what operations are performed. While performing transmission by using USRP, GNU Radio executes not only the modules directly related to transmission operation (e.g., modulation, filter), but

also other modules, such as various libraries, modules related with USRP and Python. They also have heavy complexity.

4.1.1 Transmitter Results with USRP1

GMSK:

sudo ./benchmark_tx.py -f 2400M -r 200k -m gmsk -v

Important symbols:

gr_frequency_modulator_fc::work

gr_bytes_to_syms::work

gr_interp_fir_filter_fff::work

DBPSK/DQPSK/D8PSK/QAM8/QAM16/QAM64/QAM256:

sudo ./benchmark_tx.py -f 2400M -r 200/400/600/800/1000k -m dbpsk/dqpsk/d8psk/qam8/qam16/qam64/qam256 -v

Important symbols:

get_bit_be

gr_packed_to_unpacked_bb::general_work

gr_interp_fir_filter_ccf::work

gr_diff_encoder_bb::work

gr_map_bb::work

gr_chunks_to_symbols_bc::work

Table I indicates the number of samples used by modulation specific symbols as well as the number of samples used by entire GNU Radio when that particular modulation scheme is used. Table II represents symbols used by each modulation scheme. Figure 9 provides a graphical representation of computational complexity of entire GNU Radio for each modulation scheme. Table III provides computational complexity of each major transmission function in GNU Radio.

Modulation Scheme	Samples for Important Symbols	Samples for Total Symbols
GMSK	10437	80163
DBPSK	15294	102944
DQPSK	10825	61883
D8PSK	8933	48654
QAM8	9026	48624
QAM16	8302	41815
QAM64	7279	34974
QAM256	6988	31597

Table I: Overall Transmitter Complexity for each Modulation Scheme

Symbols	GMSK	DBPSK	DQPSK	D8PSK	QAM8	QAM16	QAM64	QAM256
gr_frequency_modulator_fc	Y	N	N	N	N	N	N	N
gr_bytes_to_syms	Y	N	N	N	N	N	N	N
gr_interp_fir_filter_fff	Y	N	N	N	N	N	N	N
get_bit_be	N	Y	Y	Y	Y	Y	Y	Y
gr_packed_to_unpacked_bb	N	Y	Y	Y	Y	Y	Y	Y
gr_interp_fir_filter_ccf	N	Y	Y	Y	Y	Y	Y	Y
gr_diff_encoder_bb	N	Y	Y	Y	Y	Y	Y	Y
gr_map_bb	N	Y	Y	Y	Y	Y	Y	Y
gr_chunks_to_symbols_bc	N	Y	Y	Y	Y	Y	Y	Y

Table II: Summary of Modulation Specific Symbols for Transmitter

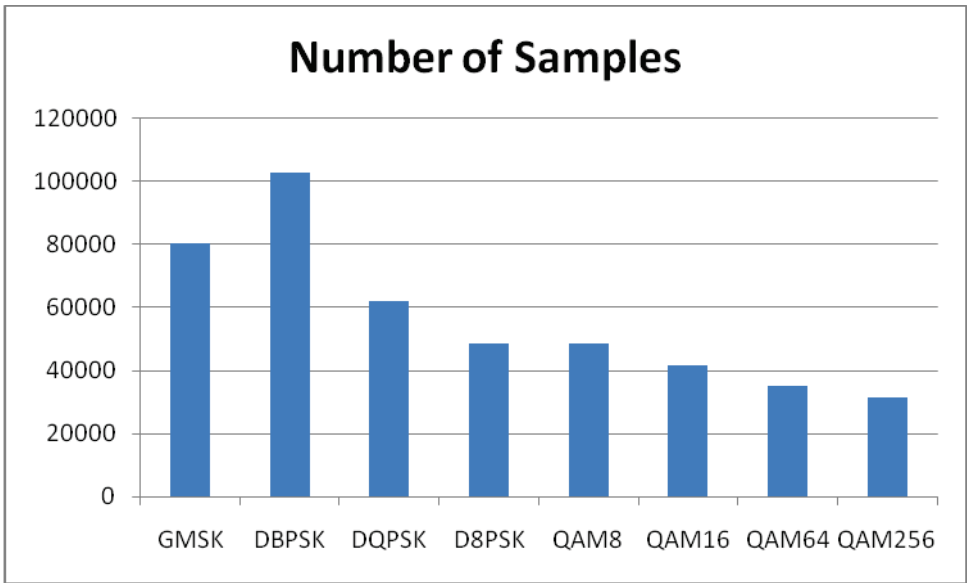

Figure 9: Transmitter Complexity for each Modulation Scheme

DBPSK	DQPSK	D8PSK	QAM8	QAM16	QAM64	QAM256	Functions
25844	12931	8784	8720	6636	4602	3368	fcomplex_dotprod_sse
19543	9728	6536	6562	4830	2942	2322	gr_fir_ccf_simd
10554	5281	3547	3534	2675	1767	1321	gr_multiply_const_cc
5204	2604	1770	1737	1303	871	654	usrp_sink_c
4415	4067	4208	4099	4045	4337	3791	get_bit_be
3932	3085	2308	2462	2484	1735	2338	gr_packed_to_unpacked_bb
3396	1673	1092	1034	754	510	411	gr_interp_fir_filter_ccf
1739	925	623	620	472	311	228	gr_diff_encoder_bb
1230	608	430	422	329	227	157	gr_map_bb
950	482	305	295	224	186	122	gr_chunks_to_symbols_bc

Table III: Computational complexity of Transmission Functions

Sequence of events in GNU Radio (irrespective of modulation scheme):

1. gr_packed_to_unpacked_bb::general_work

   get_bit_be (It is a function defined inside gr_packed_to_unpacked_bb.cc)

2. gr_map_bb::work

3. gr_diff_encoder_bb::work

4. gr_chunks_to_symbols_bc::work

5. gr_interp_fir_filter_ccf::work

6. gr_fir_ccf_simd::filter

7. fcomplex_dotprod_sse

8. gr_multiply_const_cc::work

9. usrp_sink_c::copy_to_usrp_buffer

The top 5 functions (from 1 to 5) are directly involved in modulation scheme. The bottom 4 functions from (6 to 9) are GNU Radio operations for transmission.

For DBPSK, gr_map_bb::work requires 1230 samples. This means that when it is 1 bit per symbol it requires 1230 samples. For DQPSK, it requires only 608 samples. This means that when it is 2 bits per symbol it requires 608 samples which is nothing but approximately 1/2 of the samples required by DBPSK. For D8PSK, it requires 430 samples. This means that when it is 3 bits per symbol it requires 430 samples which is nothing but approximately 1/3rd of the samples required by DBPSK. And so on for QAM256, it requires 157 samples. This means that when it is 8 bits per symbol it requires 157 samples which is nothing but approximately 1/8th of the samples required by DBPSK.

Thus, we can deduce that the number of samples required by gr_map_bb::work is directly affected by the number of bits per symbol specified by the modulation scheme. If we look closely, this result is true for all the symbols (from 2 to 5 which are related to modulation only) except for gr_packed_to_unpacked_bb.

The result does not apply to gr_packed_to_unpacked_bb::general_work and get_bit_be. This symbol is used to unpack the data from the source file into chunks of size specified by the number of bits per symbol. Now, in case of our experiments same amount of data which is 1 Megabyte is sent out. But after gr_packed_to_unpacked_bb::general_work and get_bit_be have finished their work, there would be different number of chunks of data for each modulation scheme.

For example, for DBPSK if there are 100,000 data chunks for 1 Megabyte then for DQPSK it would be 50,000 data chunks for same amount data. After gr_packed_to_unpacked_bb::general_work and get_bit_be are finished, rest of the symbols (from 2 to 5) deal with data chunks and not with the original data. As these data chunks depend on number of bits per symbol specified by modulation scheme, the number of samples required by the symbols (2 to 5) would also depend on number of bits per symbol.

This clearly implies that the computational complexity of modulation scheme itself in GNU Radio is affected by the number of bits per symbol.

However, subsequent transmission functions in GNU Radio are also affected by the selection of number of bits per symbol. This is because once the modulation of data is completed; the subsequent operations would deal with data symbols and not the original data. Now, the number of data symbols would vary from one modulation scheme to another.

Hence, the number of samples required by the symbols (from 6 to 9 which are not related to modulation but are used by GNU Radio for transimission purposes) are also

affected by the choice of modulation scheme. And thus, we get different number of samples for entire GNU Radio operation when we change the modulation scheme.

At the end, it can be said that higher the number of bits per symbol for a modulation scheme the lesser it is computationally complex. It is always preferable to use a modulation scheme with a higher number of bits per symbol if computational complexity is a priority.

As modulation changes from DBPSK to DQPSK to D8PSK, transmitter complexity decreases. It is also the case with QAM8, QAM16, QAM64 and QAM256. However, the difference in the latter is not as huge as in the former.

This is because the difference between no. of samples for a given symbol reduces from DBPSK to D8PSK and from QAM8 to QAM256. In GNU Radio, all the modulation schemes from DBPSK to all QAMs use same symbols.

As we can see for fcomplex_dotprod_sse function, DBPSK requires 25844 samples and DQPSK requires almost half of the number of samples. So the difference is huge (almost 13000 samples) in between DBPSK and DQPSK for this symbol. Now, QAM16 uses 1/4th of the samples used by DBPSK while QAM64 uses 1/6th of the samples used by DBPSK. However, the difference between samples of QAM16 and QAM64 is small (almost 2000 samples only). This can be seen in all the other symbols of a modulation scheme. Thus, it seems that the transmitter complexity reduces drastically initially and then gradually.

Fig. 10 compares the total number of cycles and the required cycles per second for each modulation scheme. As modulation changes from DBPSK to DQPSK to D8PSK, overall transmitter complexity decreases. It is also the case with QAM8, QAM16, QAM64, and QAM256. This is because the computational workload greatly depends on the number of symbols, which decreases as the modulation level increases. On the other hand, considering the different communication duration at different data rate, the required cycles per second exhibits the opposite trend, which means that lower rate communication takes more time but low voltage and frequency for the microprocessor. Due to the quadratic effect of voltage on energy, it would mean energy savings.

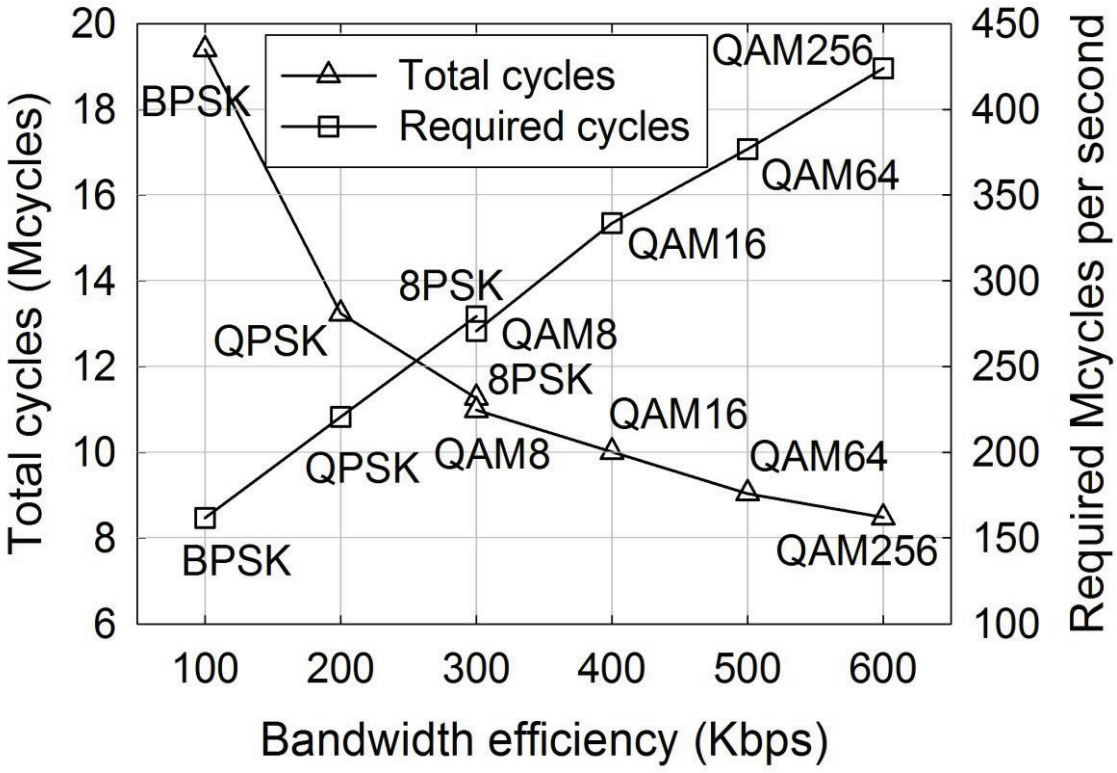

Figure 10: Computational Complexity of GNU Radio Transmitter

4.1.2 Receiver Results with USRP1

GMSK:

sudo ./benchmark_rx.py -f 2400M -r 200k -m gmsk -v

Important Symbols:

gr_fast_atan2f

gr_quadrature_demod_cf::work

gr_binary_slicer_fb::work

gr_clock_recovery_mm_ff::general_work

gr_clock_recovery_mm_ff::forecast

gri_mmse_fir_interpolator::interpolate

gri_mmse_fir_interpolator::ntaps

DBPSK/DQPSK:

sudo ./benchmark_rx.py -f 2400M -r 200/400k -m dbpsk/dqpsk -v

Important Symbols:

gr_feedforward_agc_cc::work

gr_constellation_decoder_cb::work

gr_multiply_const_cc::work

gr_unpack_k_bits_bb::work

gr_diff_phasor_cc::work

gr_map_bb::work

gr_interp_fir_filter_ccf::work

gr_sincosf

gri_mmse_fir_interpolator_cc::interpolate

gri_mmse_fir_interpolator_cc::ntaps()

gr_mpsk_receiver_cc::mm_sampler

gr_mpsk_receiver_cc::mm_error_tracking

gr_mpsk_receiver_cc::general_work

gr_mpsk_receiver_cc::phase_error_tracking

gr_mpsk_receiver_cc::decision_bpsk/qpsk

gr_mpsk_receiver_cc::phase_error_detector_bpsk/qpsk

gr_mpsk_receiver_cc::forecast

Modulation	Samples for Important Symbols	Samples for All Symbols
GMSK	52634	181274
DBPSK	265033	484742
DQPSK	144508	285547

Table IV: Overall Receiver Complexity for each Modulation Scheme

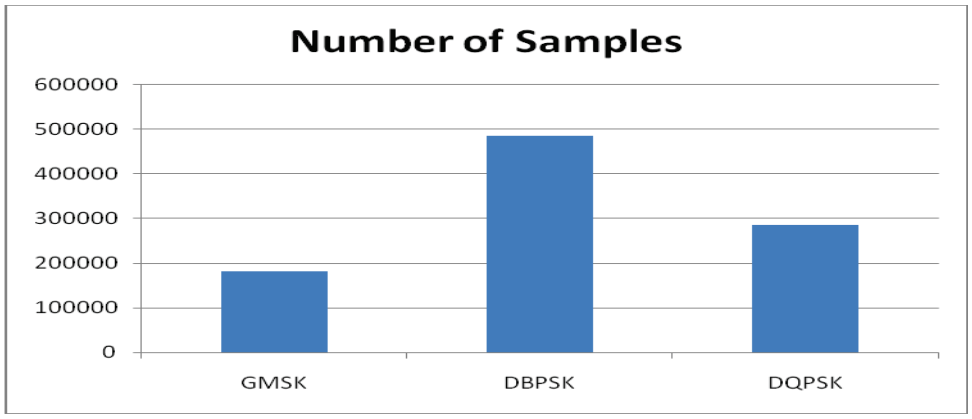

Figure 11: Receiver Complexity for each Modulation Scheme

DBPSK	DQPSK	Symbols
160234	82251	gr_feedforward_agc_cc::work
60503	28913	fcomplex_dotprod_sse
21075	10346	gr_mpsk_receiver_cc::mm_sampler
20453	10474	gr_fft_filter_ccc::work
18647	8288	gr_mpsk_receiver_cc::mm_error_tracking
13827	7449	gr_mpsk_receiver_cc::general_work
13641	15796	gr_correlate_access_code_bb::work
13318	7372	gr_fir_ccf_simd::filter
12629	10575	gr_constellation_decoder_cb::work
8281	4788	gr_mpsk_receiver_cc::phase_error_tracking
7285	5518	gr_count_bits32
5443	2819	gr_single_pole_iir<double, double, double>::filter
5377	2770	gr_multiply_const_cc::work
4156	2786	gr_unpack_k_bits_bb::work
4086	3110	gr_interp_fir_filter_ccf::work
4002	4146	gr_count_bits64
3844	1938	gr_probe_avg_mag_sqrd_c::work
3728	1885	gri_mmse_fir_interpolator_cc::interpolate
3546	1795	gr_diff_phasor_cc::work
3353	2026	gr_sincosf
2998	3044	gr_framer_sink_1::work
2961	1543	usrp_source_c::copy_from_usrp_buffer
2629	1347	gr_mpsk_receiver_cc::phase_error_detector_bpsk/qpsk
2324	3412	gr_mpsk_receiver_cc::decision_bpsk/qpsk
1298	688	gr_map_bb::work
940	476	gri_fft_complex::execute
285	169	gr_mpsk_receiver_cc::forecast
11	6	gri_mmse_fir_interpolator_cc::ntaps

Table V: Complexity of Individual Reception Symbols in GNU Radio

Symbols	GMSK	DBPSK	DQPSK
gr_fast_atan2f	Y	N	N
gr_quadrature_demod_cf	Y	N	N
gr_binary_slicer_fb	Y	N	N
gr_clock_recovery_mm_ff	Y	N	N
gri_mmse_fir_interpolator	Y	N	N
gr_feedforward_agc_cc::work	N	Y	Y
gr_mpsk_receiver_cc	N	Y	Y
gr_constellation_decoder_cb	N	Y	Y
gr_multiply_const_cc	N	Y	Y
gr_unpack_k_bits_bb	N	Y	Y
gr_diff_phasor_cc	N	Y	Y
gr_map_bb	N	Y	Y
gr_sincosf	N	Y	Y
gri_mmse_fir_interpolator_cc	N	Y	Y

Table VI: Summary of Modulation Specific Symbols for Receiver

Table IV indicates the number of samples used by modulation specific symbols as well as the number of samples used by entire GNU Radio for reception when that particular modulation scheme is used. Table V provides computational complexity of each major reception function in GNU Radio. Figure 11 provides a graphical representation of computational complexity of entire GNU Radio for each modulation scheme for reception. Table VI represents symbols used by each modulation scheme.

As seen in transmitter, most of the symbols associated with DQPSK require approximately half the number of samples than that required in DBPSK. Also, some of the symbols which are not associated with modulation and are used only by GNU Radio require half the number of samples when using DQPSK than that required when using DBPSK. So, we can see even in case of receiver, number of bits per symbol not only affects the number of samples required by modulation functions but also the GNU Radio functions.

However, there are four symbols related to modulation which are gr_constellation_decoder_cb, gr_unpack_k_bits_bb::work, gr_interp_fir_filter_ccf::work, and gr_mpsk_receiver_cc::decision_qpsk/bpsk which require more or less number of samples irrespective of whether the modulation scheme is DBPSK or DQPSK. It is possibly because the input parameter for these functions is not just the number of data symbols received but they depend on some other parameters too.

4.1.3 Complexity Analysis with Bandwidth Variation

Next, we observed transmission complexity with varying bandwidth. We created a bandwidth effect by changing bit rate. For example, we performed transmission and reception with DBPSK 200kbps (200KHz) and DBPSK 400kbps (400KHz) and also with DQPSK 400kbps (200KHz) and DQPSK 800kbps (400 KHz).

DBPSK:

Bits per symbol = 1

Samples per symbol = 2

Data Rate = 200kbps

Sampling Frequency or Rate = 2 Samples/Symbol * 1 Symbol/Bit * 200k Bits/Second

= 400k Samples/Second

Bandwidth = Sampling Frequency/2 = 200kHz

Data Rate = 400kbps

Sampling Frequency or Rate = 2 Samples/Symbol * 1 Symbol/Bit * 400k Bits/Second

= 800k Samples/Second

Bandwidth = Sampling Frequency/2 = 400kHz

DQPSK:

Bits per symbol = 2

Samples per symbol = 2

Data Rate = 400kbps

Sampling Frequency or Rate = 2 Samples/Symbol * (1/2) Symbol/Bit * 400k Bits/Second

= 400k Samples/Second

Bandwidth = Sampling Frequency/2 = 200kHz

Data Rate = 800kbps

Sampling Frequency or Rate = 2 Samples/Symbol * (1/2) Symbol/Bit * 800k Bits/Second

= 800k Samples/Second

Bandwidth = Sampling Frequency/2 = 400kHz

Modulation Scheme	Transmitter		Receiver	
	200KHz	400KHz	200KHz	400KHz
DBPSK	15294	15378	265033	303578
DQPSK	10825	10660	144508	181416

Table VII: Modulation-specific Complexity Analysis with Bandwidth Variation

Modulation Scheme	Transmitter		Receiver	
	200KHz	400KHz	200KHz	400KHz
DBPSK	102944	102662	484742	540836
DQPSK	61883	61850	285547	359396

Table VIII: Overall GNU Radio Complexity Analysis with Bandwidth Variation

Table VII provides complexity analysis for DBPSK and DQPSK related-only symbols for both transmitter and receiver with bandwidth variation. Table VIII provides complexity analysis for entire GNU Radio transmitter and receiver when DBPSK and DQPSK are used with bandwidth variation.

Comparing two results, we found that transmitter complexity is almost constant while bit rate and bandwidth is doubled. Also, this is similarly observed with DQPSK. This reason is that the number of instructions is not changed even though bandwidth increases. However, unlike the transmitter complexity, the receiver complexity increases around 11% and 25% for DBPSK and DQPSK, respectively. This is because high bitrate increases sampling rate to receive incoming data from the channel. We observed that, in case of transmitter, the number of samples (instructions) required to write all the symbols into GNU Radio USRP buffer (usrp sink c - Interface to Universal Software Radio Peripheral Tx path) is same in both the cases (200KHz and 400KHz). However, in case of receiver, the GNU Radio USRP buffer (usrp_source_c - Interface to Universal

Software Radio Peripheral Rx path), the number of samples increases for 400KHz scenario then 200KHz.

4.2 BBN 802.11b-based Complexity Analysis

USRP/GNU Radio-based experiment mentioned above is not sufficient because it does not implement the 802.11 standard. To obtain more reliable results, we profiled transmission complexity of BBN 802.11b implementation in GNU Radio [3]. We use Oprofile [20] again. Note that the BBN 802.11b implementation does not include a transmitter with 5.5Mbps and 11Mbps data rates. In 802.11b, CCK (complementary code keying) encoding is based on differential QPSK modulation to encode the phase parameters which are used to make 8-bit CCK code words. Based on this, we implemented a block for the purpose of profiling computational complexity of CCK modulation. Fig. 12 provides signal flow in BBN 802.11b transceiver. Table IX shows detailed information on the profiled results for a subset of symbols. It is interesting to observe that, with a few exceptions, each symbol block takes a decreasing amount of computations as data rate increases.

Figure 13 shows a similar trend as it was observed in the GNU Radio/USRP transmitter complexity for each modulation scheme. As we see here, BPSK requires higher number of total cycles than QPSK but lower number of cycles per second due to different communication data rates. Same trend is observed for 4-CCK and 8-CCK.

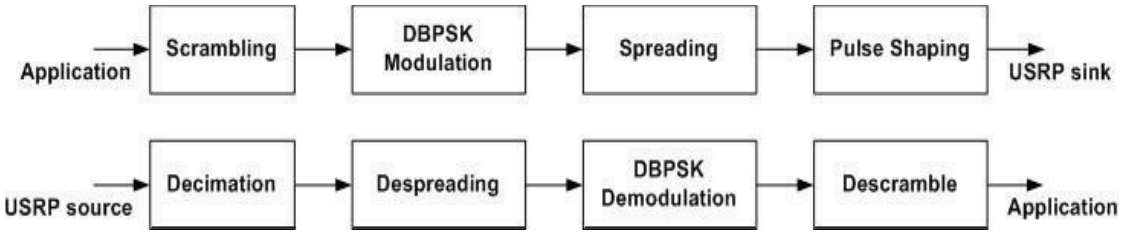

Figure 12: BBN 802.11b Transceiver

Symbols	1Mbps	2Mbps	5.5Mbps	11Mbps
bbn_scrambler_bits	0.65	0.65	0.67	0.67
gr_packed_to_unpacked	0.24	0.18	0.17	0.17
get_bit_be	0.14	0.19	0.15	0.15
gr_fir_ccf_simd::filter	3.36	1.71	0.79	0.41
fcomplex_dotprod_sse	3.11	1.48	0.80	0.37

Table IX: Profiling results of BBN 802.11b Transmitter

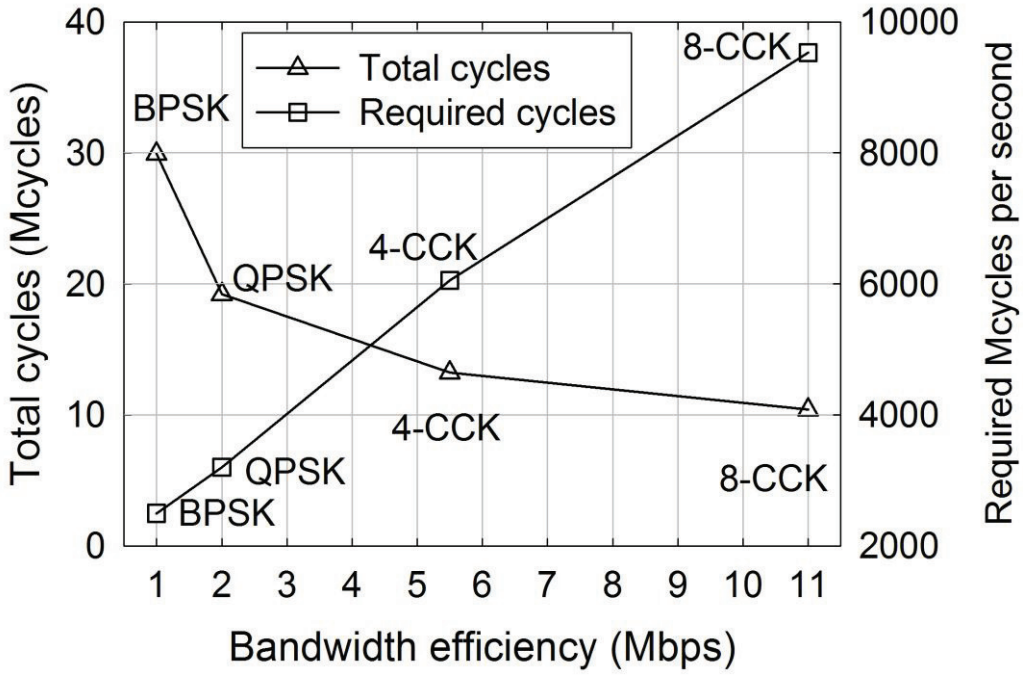

Figure 13: Computational Complexity of BBN 802.11b Transmitter

4.3 Matlab-based Complexity Analysis

Using Matlab and Simulink, one can design SDR in a modularized manner. Moreover, they allow simulation and performance analysis of SDRs [22]. We additionally use the Matlab implementation of 802.11b to estimate the computational complexity of SDR functions. Here are the details of our experiment. (i) We use Matlab V7.9.0.529, Simulink V7.4, Communications Toolbox V4.4, Signal Processing Toolbox V6.12, Communications Blockset V4.3, and Signal Processing Blockset V6.10 on Ubuntu 8.04 (Hardy). (ii) We use again Oprofile (0.9.6) for profiling the computations. (iii) The packet size in each scenario is 1024 bytes and 1000 packets are transmitted for each data rate. (iv) The PLCP header size is 192 bits (long preamble) and 128 bits (short preamble). Fig. 14, 15 and 16 show SDR implementation in Matlab and Simulink. Fig. 17 shows the total number of cycles at each data rate as well as the corresponding cycles per second. It shows that as the data rate increases the computational complexity decreases which is consistent with the results that we obtained using USRP/GNU Radio and BBN 802.11b.

Figure 14: Matlab SDR

Figure 15: Matlab SDR Transmitter

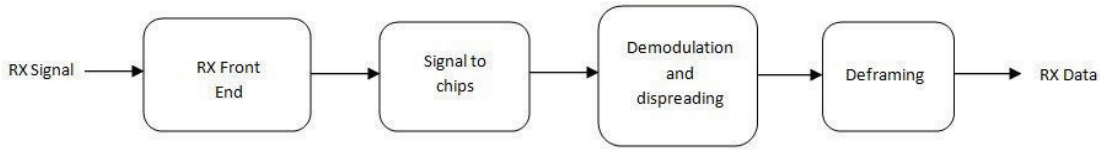

Figure 16: Matlab SDR Receiver

The Oprofile profiling results provided a huge number of functions used by Matlab and Simulink for simulation of SDR. However, there are few major functions that are required for signal processing purposes while the rest of the functions are used for performing mathematical functions by Matlab. Table X provides a list of all the major functions of Matlab and Simulink and also the total number of samples used by each function. Figure 17 shows similar trend for computational complexity for Matlab SDR as seen in the case of GNU Radio/USRP and BBN 802.11b transceiver.

	Functions	1 Mbps	2 Mbps		5.5 Mbps		11 Mbps	
			Long Preamble	Short Preamble	Long Preamble	Short Preamble	Long Preamble	Short Preamble
1	sdspfilter2	1355325	693185	677675	271830	256340	151445	135940
2	sdspupfir2	347730	177850	173855	69740	65765	38855	34875
3	Sdspstatfcns	94100	48600	47400	18800	17100	10275	8960
4	scomawgnchan2	15585	7970	7800	3125	2955	1745	1555
5	sdspdsamp2	9325	4675	4575	1765	1745	920	900
6	Scomapskdemod	3575	2470	2460	1185	1200	645	625
7	Scomapskmod	1460	1130	1115	490	470	965	940
8	scomerrrate2	465	465	455	460	465	460	455
9	Scominttobit	185	180	182	190	185	185	195
10	Sdspstatminmax	N/A	N/A	N/A	440	450	3095	3090
11	sdspperm2	N/A	N/A	N/A	130	140	55	62
	TOTAL	1827750	936525	915517	368155	346815	208645	187597
	TOTAL (Entire MATLAB)	2123500	1091000	1071500	458750	437000	299500	280000

Table X: Computational Complexity for Individual Symbols in Matlab

As seen above there are quite a few functions used for signal processing by Matlab. Each function has its own significance and is described below briefly:

1. **sdspfilter2** - RX Pulse Shaping filter (Direct form II Transpose filter) in RX Front End block.

2. **sdspupfir2** - TX Pulse shaping filter for FIR Interpolation in TX Upsampling and Pulse shaping block.

3. **scomawgnchan2** - AWGN Channel block.

4. **sdspdsamp2** - Used in RX Signal to Chips conversion block.

5. **scomapskmod** - Used in TX Modulation and Spreading block.

6. **scomapskdemod** - Used in RX Demodulation and Despreading block.

7. **scominttobit** - Converting random data source bytes into bits on Transmitter side between Data source and Framing block.

8. **sdspstatfcns** - Used in RX Demodulation and Despreading block to pick out maximum value over a set of input elements.

9. **sdspperm2** - Used in RX Demodulation and Despreading block to select or reorder a set of input elements.

10. **sdspstatfcns -** Statistical function (using variance) to compute TX signal power.

11. **scomerrrate2** – Used for error rate calculation for BER purposes.

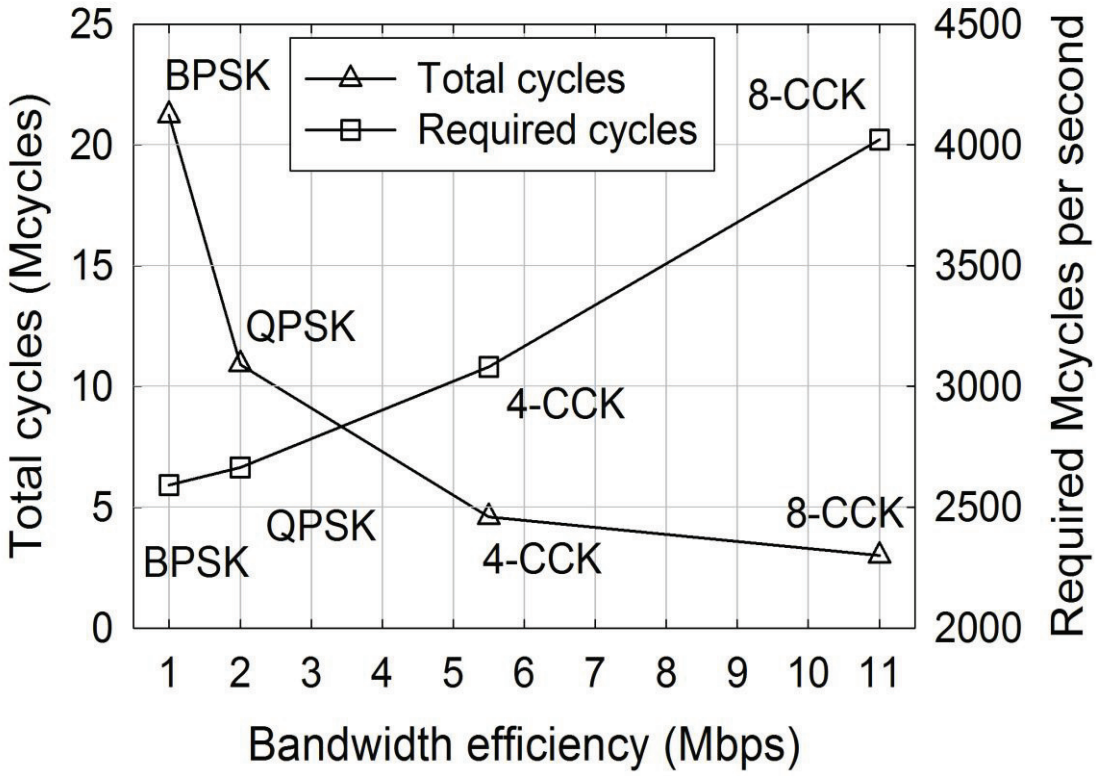

Figure 17: Computational Complexity for Matlab SDR

4.4 Summary

In case of transmitter, we see that the number of samples required by a particular modulation scheme as well as by entire GNU Radio goes down as the number of bits per symbol increases. It is clearly observed that the overall number of samples go down from DBPSK to D8PSK as well as from QAM8 to QAM256. However, for different modulation scheme, the duration of communication is different as it can support different data rate. Thus, we see that the communication duration goes down from DBPSK to

D8PSK and from QAM8 to QAM256. As a result, the number of samples per second show opposite trend than the overall number of samples required by respective modulation schemes. Thus, number of samples per second increases as the number of bits per symbol increases i.e. from DBPSK to D8PSK and QAM8 to QAM256.

In case of receiver, very similar trend as seen in transmitter is observed. We see that the number of samples required only by DQPSK is less than that of DBPSK. Moreover, the number of samples used by entire GNU Radio while using DQPSK is also less than that of DBPSK. We have also observed similar results in case of BBN 802.11b and Matlab SDR. Overall, we can say that DQPSK is computationally less intensive than DBPSK to communicate same amount of data.

As a summary, highly sophisticated modulation schemes are preferable as they deliver messages faster as well as execute small number of instructions. However, highly sophisticated modulation schemes have high BER and hence, the performance obtained is slightly at the expense of reliability. Also, as we observed, higher modulation schemes will execute higher number of instructions per second. Hence, the microprocessor running SDR will need higher voltage and frequency at that particular instant which results in higher instantaneous power consumption. Thus, this complexity analysis can be useful to choose desired modulation scheme based on the application's performance requirements as well as available power resources.

CHAPTER V

CONCLUSION AND FUTURE WORK

The main aim of this thesis was to analyze computational complexity of different signal processing functions utilized in software defined radio on different platforms. This analysis in future would then help to devise a SDR-based communication system which provides optimal performance with minimal power consumption. In this thesis, we were able to analyze different modulation schemes such as GMSK, M-DPSK, and QAMs. We also analyzed the performance of IEEE 802.11b standard. In this thesis, we were able to realize that the computational complexity of any signal processing function heavily depends on the number of bits per symbol (constellation size) for a particular modulation scheme. The energy and delay performance can be traded off against each other by varying constellation size or by changing the modulation scheme.

The future goal of this research should be to expand this work to many other modulation schemes as well as other popular standards such as IEEE 802.11a/g/n, Bluetooth and ZigBee protocols. Based on all this analysis, we can devise mechanisms to configure software radio on the fly to meet the application requirements along with low power consumption and efficient performance.

The computational complexity analysis could be useful to design a communication system which uses both modulation scaling and dynamic voltage scaling for high performance and low power consumption. In SDR-based wireless systems, different modulation schemes or data rates demand different computational workload, thus making it possible to save energy by applying the DVS technique as in conventional energy-aware processor design.

BIBLIOGRAPHY

[1] K. Tan, J. Zhang, H. Wu, F. Ji, H. Liu, Y. Ye, S. Wang, Y. Zhang, W. Wang, and G. M. Voelker, "*Sora: High Performance Software Radio Using General Purpose Multi-core Processors,*" Proc. ACM/USENIX NSDI, 2009.

[2] V. Raghunathan, C. Schurgers, S. Park, and M.B. Srivastava, "*Energy-aware wireless microsensor networks,*" IEEE Signal Processing Magazine, vol. 19, pages 40-50, March, 2002.

[3] A. Y. Wang, S. H. Cho, C. G. Sodini, and A. P. Chandrakasan, "*Energy efficient modulation and MAC for asymmetric RF microsensor systems*," Proc. ACM ISLPED, 2001.

[4] P. G. M. Baltus and R. Dekker, "*Optimizing RF Front Ends for Low Power,*" Proceedings of the IEEE, Vol. 88, No. 10, October, 2000, 1546-1559.

[5] B. A. Myers, J. B. Willingham, P. Landy, M. A. Webster, P. Frogge, and M. Fischer, "*Design Considerations for Minimal-Power Wireless Spread Spectrum Circuits and Systems,*" Proceedings of the IEEE, Vol. 88, No. 10, October, 2000, 1598-1612.

[6] A. Kamerman, and L. Monteban, "*WaveLAN-II: A High-performance Wireless LAN for the Unlicensed Band,*" Bell Labs Technical Journal, Summer, 1997, 118-133.

[7] S. Cui, R. Madan, A. Goldsmith and S. Lall, "*Energy-Delay Tradeoffs for Data Collection in TDMA-based Sensor Networks,*" Proc. IEEE ICC, 2005.

[8] C. Schurgers, V. Raghunathan, and M. B. Srivastava, "*Power management for energy-aware communication systems,*" ACM Trans. On Embedded Computing Systems, Vol. 2, No. 3, 2003, 431-447.

[9] S. Cui, A. Goldsmith, A. Bahai, "*Energy-constrained Modulation Optimization,*" IEEE Trans. on Wireless Communications, September, 2005.

[10] R. Min, and A. Chandrakasan, "*A framework for energy-scalable communication in high-density wireless networks,*" Proc. ACM ISLPED, 2002.

[11] Y. Lin, Y. L. Hyunseok, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, "*SODA: A Low-power Architecture For Software Radio,*" ACM/IEEE ISCA, 2006.

[12] E. Marpanaji, B. Riyanto, A. Z. R. Langi, A. Kurniawan, A. Mahendra and T. Liung, "*Experimental Study of DQPSK Modulation on SDR Platform,*" ITB Journal, Vol. 1, No.2, 2007.

[13] *www.gnuradio.org*

[14] S. Hirve, "*Multihop Transmission Opprotunistic Protocol on Software Radio,*" Master Thesis, Cleveland State University, Summer, 2009.

[15] *www.ettus.com*

[16] T. Shen, "*Experimental Study of Multirate Margin in Software Defined Multirate Radio,*" Master Thesis, Cleveland State University, Fall, 2009.

[17] F. A. Hamza, "*The USRP under 1.5X Magnifying Lens!,*" June 2008. Available at: *gnuradio.org/redmine/attachments/download/129*

[18] *www.mathworks.com*

[19] *www.ibm.com/developerworks/linux/library/l-oprof.html*

[20] *www.oprofile.sourceforge.net*

[21] K. A. Jamieson, "*The SoftPHY Abstraction: from Packets to Symbols in Wireless Network Design,*" Ph.D. Dissertation, MIT, June 2008.

[22] R. D. Raut, K. D. Kulat, "*BER performance maintenance at high data rates in cognitive radio,*" Proc. Int'l Conf. Electronics, Communications and Computer (CONIELECOMP), 2010.

APPENDICES

APPENDIX A

OPROFILE TOOLS

OProfile Tools [20]:

Ophelp – Lists available events supported by the processor along with their short descriptions.

Opcontrol – Tool that allows the user to configure different parameters for profiling and data collection.

Opreport – Retrieves useful profile data and generates reports based on user specifications.

Opannotate – OProfile users to produce reports with annotations of source or assembly code so that it becomes easier for the user to identify where exactly the problem is. But the user has to make sure that it enables profiling with debugging symbols.

Opgprof – Provides gprof-style data files for binary that can be used with gprof.

Oparchive – This tool will collect all the data collected by OProfile and will save it in an archive. This archive can then be easily transferred from one machine to another based on the requirements of the user.

Opimport – This tool can be used by the user who has moved the data collected from the machine which was used for profiling to some other machine. It will help the user to convert the original file into a format supported by current host machine.

APPENDIX B

GNURADIO SIGNAL PROCESSING FUNCTIONS

Transmitter Functions:

GMSK:

sudo ./benchmark_tx.py -f 2400M -r 200k -m gmsk -v

Important functions:

gr_frequency_modulator_fc::work

gr_bytes_to_syms::work

gr_interp_fir_filter_fff::work

DBPSK:

sudo ./benchmark_tx.py -f 2400M -r 200k -m dbpsk -v

Important functions:

get_bit_be

gr_packed_to_unpacked_bb::general_work

gr_interp_fir_filter_ccf::work

gr_diff_encoder_bb::work

gr_map_bb::work

gr_chunks_to_symbols_bc::work

DQPSK:

sudo ./benchmark_tx.py -f 2400M -r 400k -m dqpsk -v

Important functions:

get_bit_be

gr_packed_to_unpacked_bb::general_work

gr_interp_fir_filter_ccf::work

gr_diff_encoder_bb::work

gr_map_bb::work

gr_chunks_to_symbols_bc::work

D8PSK:

sudo ./benchmark_tx.py -f 2400M -r 600k -m d8psk -v

Important functions:

get_bit_be

gr_packed_to_unpacked_bb::general_work

gr_interp_fir_filter_ccf::work

gr_diff_encoder_bb::work

gr_map_bb::work

gr_chunks_to_symbols_bc::work

QAM8:

sudo ./benchmark_tx.py -f 2400M -r 600k -m qam8 -v

Important functions:

get_bit_be

gr_packed_to_unpacked_bb::general_work

gr_interp_fir_filter_ccf::work

gr_diff_encoder_bb::work

gr_map_bb::work

gr_chunks_to_symbols_bc::work

QAM16:

sudo ./benchmark_tx.py -f 2400M -r 800k -m qam16 -v

Important functions:

get_bit_be

gr_packed_to_unpacked_bb::general_work

gr_interp_fir_filter_ccf::work

gr_diff_encoder_bb::work

gr_map_bb::work

gr_chunks_to_symbols_bc::work

QAM64:

sudo ./benchmark_tx.py -f 2400M -r 1000k -m qam64 -v

Important functions:

get_bit_be

gr_packed_to_unpacked_bb::general_work

gr_interp_fir_filter_ccf::work

gr_diff_encoder_bb::work

gr_map_bb::work

gr_chunks_to_symbols_bc::work

QAM256:

sudo ./benchmark_tx.py -f 2400M -r 1000k -m qam256 -v

Important functions:

get_bit_be

gr_packed_to_unpacked_bb::general_work

gr_interp_fir_filter_ccf::work

gr_diff_encoder_bb::work

gr_map_bb::work

gr_chunks_to_symbols_bc::work

Transmitter Functions [13]:

gr_frequency_modulator_fc::work – It is a Frequency modulator block which accepts float type input and gives complex baseband output. In frequency modulation, changes in the baseband signal are imposed on the frequency of the carrier wave.

gr_interp_fir_filter_fff::work – This block performs interpolation with FIR filters. The input and output as well as taps used for interpolation for this block are in float type.

gr_bytes_to_syms::work – This block is used for converting byte streams to symbol stream. The input for this block is byte steam while the output is float stream.

gr_packed_to_unpacked_bb::general_work – This block is used for converting packed bytes stream to unpacked bytes stream. The input as well as output for this block is a unsigned characters stream.

gr_interp_fir_filter_ccf::work - This block performs interpolation with FIR filters. The input and output for this block are of gr_complex type while taps are of float type.

gr_diff_encoder_bb::work - This block implements differential encoder (b[0] = (a[0] + b[-1]) % M). It operates on bits.

gr_map_bb::work – This block maps input bit pattern to a pre-defined bit pattern (output[i] = map[input[i]]). This block also operates on bits.

gr_chunks_to_symbols_bc::work – This block produces a float stream in Z dimensions from unpacked bytes stream. So the output is stream of gr_complex while the input is unsigned characters stream.

output[n Z + m] = symbol_table[input[n] Z + m], m=0,1,...,Z-1

Here, Z is dimensions and its value is 1 by default.

This block along with the gr_packed_to_unpacked and gr_chunks_to_symbols are used for converting bytes into complex symbols.

gr_multiply_const_cc::work - Output = Input * Constant

usrp_sink_c::copy_to_usrp_buffer – This block provides interface for GNU Radio to Universal Software Radio Peripheral (USRP) Tx path. Input for this block is gr_complex.

gr_fir_ccf_simd::filter – It is a block which implements the SIMD model for gr_fir_ccf. It helps in handling problems related with SSE and 3DNOW subclasses. gr_fir_ccf takes complex symbols as input and provides complex symbols as output. It uses float taps.

Receiver Functions:

GMSK:

sudo ./benchmark_rx.py -f 2400M -r 200k -m gmsk -v

Important Symbols:

gr_fast_atan2f

gr_quadrature_demod_cf::work

gr_binary_slicer_fb::work

gr_clock_recovery_mm_ff::general_work

gr_clock_recovery_mm_ff::forecast

gri_mmse_fir_interpolator::interpolate

gri_mmse_fir_interpolator::ntaps

DBPSK:

sudo ./benchmark_rx.py -f 2400M -r 200k -m dbpsk -v

Important Symbols:

gr_feedforward_agc_cc::work

gr_constellation_decoder_cb::work

gr_multiply_const_cc::work

gr_unpack_k_bits_bb::work

gr_diff_phasor_cc::work

gr_map_bb::work

gr_interp_fir_filter_ccf::work

gr_sincosf

gri_mmse_fir_interpolator_cc::interpolate

gri_mmse_fir_interpolator_cc::ntaps()

gr_mpsk_receiver_cc::mm_sampler

gr_mpsk_receiver_cc::mm_error_tracking

gr_mpsk_receiver_cc::general_work

gr_mpsk_receiver_cc::phase_error_tracking

gr_mpsk_receiver_cc::decision_bpsk

gr_mpsk_receiver_cc::phase_error_detector_bpsk

gr_mpsk_receiver_cc::forecast

DQPSK:

sudo ./benchmark_rx.py -f 2400M -r 400k -m dqpsk -v

Important Symbols:

gr_feedforward_agc_cc::work

gr_constellation_decoder_cb::work

gr_multiply_const_cc::work

gr_unpack_k_bits_bb::work

gr_diff_phasor_cc::work

gr_map_bb::work

gr_interp_fir_filter_ccf::work

gr_sincosf

gri_mmse_fir_interpolator_cc::interpolate

gri_mmse_fir_interpolator_cc::ntaps()

gr_mpsk_receiver_cc::mm_sampler

gr_mpsk_receiver_cc::mm_error_tracking

gr_mpsk_receiver_cc::general_work

gr_mpsk_receiver_cc::phase_error_tracking

gr_mpsk_receiver_cc::decision_bpsk

gr_mpsk_receiver_cc::phase_error_detector_bpsk

gr_mpsk_receiver_cc::forecast

Receiver Functions [13]:

gr_fast_atan2f – This function implements Fast arc tangent using table lookup and linear interpolation.

gr_quadrature_demod_cf::work – This block implements quadrature demodulator. Quadrature demodulator is used in frequency modulation, frequency shift keying and Gaussian minimum shift keying. The input is complex baseband and the output is of float type.

gr_binary_slicer_fb::work – This function slices float binary symbol providing 1 bit as an output. If x < 0 then 0 and if x >= 0 then 1.

gr_clock_recovery_mm_ff::general_work – This function uses the Mueller and Müller (M&M) implementation for discrete-time error-tracking synchronizer. It operates on float input and output.

gri_mmse_fir_interpolator::interpolate – This block is used to compute samples between n(m*Ts) signal samples.

It uses a Mininum Mean Squared Error interpolator. It is better suited for signals that has the bandwidth around 1/(4*Ts). Ts is the duration between two samples.

In this case, mu is quantized to the 32nd's of a sample. It is a fractional delay and is represented as float. It is always in the range of [0, 1].

This function provides the output as a single value of interpolation of input value. However it is necessary to have ntaps valid entries. All the input values from 0 to ntaps-1 are used as reference to compute the output.

gr_feedforward_agc_cc::work – This block uses non-causal AGC. It computes the gain that will be required by receiver by analyzing a pre-determined number of input samples. The input and output for this function are both of gr_complex type.

gr_constellation_decoder_cb::work – This block implements Constellation Decoder. The input is gr_complex while output is bits.

gr_unpack_k_bits_bb::work – It converts the incoming byte with n bits into n output bytes with each bit located in the LSB of the output byte.

gr_diff_phasor_cc::work - This block implements differential decoder.

gr_mpsk_receiver_cc – This block uses phase, frequency, and symbol synchronization for receiving M-ary PSK signals.

It locks carrier frequency and phase in order to receive signals. It also performs symbol timing recovery. Currently it can be used for DBPSK, DQPSK and D8PSK. It is assumed that it can also demodulate OQPSK and PI/4 DQPSK modulated signals.

Costas loop are used for synchronizing phase and frequency of the incoming signals. They perform error check in the incoming signal by comparing it to the nearest constellation point. Based on the output of the Costas loop, the phase and frequency of the NCO are modified. This block already has optimized phase detection scheme implemented for BPSK and QPSK. In case of 8PSK, it uses brute force computation.

Modified Mueller and Muller circuit is used for symbol synchronization.

The modified circuit is used to reduce the noise. It interpolates a sample from every mu samples using the NCO. It finds the sampling error by analyzing earlier symbols.

APPENDIX C

MISCELLANEOUS COMMANDS

Transmitter profiling using benchmark_tx.py:

sudo ./benchmark_tx.py -f 2400M -m MOD_SCHEME -r DATA_RATE -v

This command is used to run benchmark_tx.py with desired input options.

MOD_SCHEME = DBPSK, DQPSK, D8PSK, GMSK, QAM8, QAM16, QAM64 and QAM256.

DATA_RATE = Data rate is selected based on modulation scheme.

For example, DQPSK will have higher data rate as compared to DBPSK

Receiver profiling using benchmark_rx.py:

sudo ./benchmark_rx.py -f 2400M -m MOD_SCHEME -r DATA_RATE -v

MOD_SCHEME = DBPSK, DQPSK, GMSK.

DATA_RATE = Set same as that of transmitter.

Note: I was not able to receive (decode) any packets correctly using D8PSK. Moreover, the demodulator block for QAM8, QAM16, QAM64 and QAM256 are not yet available in GNU Radio package.

OProfile commands:

All the commands for OProfile are executed in a separate terminal tab.

First of all setup all the parameters for OProfile. Following are the parameters that I am using currently in OProfile:

Event 0: INSTR_RETIRED:50000:1:1:1

Separate options: library

vmlinux file: none

Image filter: none

Call-graph depth: 0

Next is to setup a folder where we want to store the profiling results.

sudo opcontrol --session-dir=PATH_TO_FOLDER

Next, I execute benchmark_tx.py and benchmark_rx.py in a separate tab as shown above.

Then, I run a script file to start the OProfile tool. The command:

source start.sh

Once, the execution of benchmark_tx.py and benchmark_rx.py is finished. I run another script file to stop the OProfile. The command is:

source stop.sh

Next, is to use opreport to generate text file of the results. The commands are as follows:

sudo opreport --session-dir=PATH_TO_FOLDER > FILE_NAME

sudo opreport -l --session-dir=PATH_TO_FOLDER > FILE_NAME

sudo opreport -d --session-dir=PATH_TO_FOLDER > FILE_NAME

The results are stored in the FILE_NAME. The -l and -d provide detailed profiling results.