Performance Benchmarking Software-Defined Radio Frameworks: GNURadio and CRTSv.2

Kalyani S. Gadgil

Thesis submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science in
Computer Engineering

Carl B. Dietrich, Chair
Nicholas F. Polys
Walid Saad
Allen B. MacKenzie

February 21, 2020
Blacksburg, Virginia

Keywords: Software Radios, Performance
Copyright 2020, Kalyani S. Gadgil
In this thesis, we benchmark the Cognitive Radios Test System version 2.0 (CRTSv.2) to analyze its software performance with respect to its internal structure and design choices. With the help of system monitoring and profiling tools, CRTSv.2 is tested to quantitatively evaluate its features and understand its shortcomings. With the help of GNU Radio, a popular, easy-to-use software radios framework, we ascertain that CRTSv.2 has a low memory footprint, fewer dependencies and overall, is a lightweight framework that can potentially be used for real-time signal processing. Several open-source measurement tools such as valgrind, perf, top, etc. are used to evaluate the CPU utilization, memory footprint and to postulate the origins of latencies. Based on our evaluation, we observe that CRTSv.2 shows a CPU utilization of approximately 9% whereas GNU Radio is 59%. CRTSv.2 has lower heap memory consumption of approximately 3MB to GNU Radio’s 25MB. This study establishes a methodology to evaluate the performance of two SDR frameworks systematically and quantitatively.
When picking the best person for the job, we rely on the person’s performance in past projects of a similar nature. The same can be said for software. Software radios provide the capability to perform signal processing functions in software, making them prime candidates towards solving modern problems such as spectrum scarcity, internet-of-things(IoT) adoption, vehicle-to-vehicle communication etc. In order to operate and configure software radios, software frameworks are provided that let the user make changes to the waveform, perform signal processing and data management. In this thesis, we consider two such frameworks, GNU Radio and CRTSv.2. A software performance evaluation is conducted to assess framework overheads contributing to operation of an orthogonal frequency-division multiplexing (OFDM) digital modulation scheme. This provides a quantitative analysis of a signals-specific use case which can be used by researchers to evaluate the optimal framework for research. This analysis can be generalized for different signal processing capabilities by understanding the total framework overhead removed from signal processing costs.
Dedication

I dedicate this thesis to my best friends, Monica, Bella, Twenty-five, Fifi and Sarang.

Thank you for hours of love, patience and coffee.
Acknowledgments

I would like to thank Dr. Lance Arsenault for CRTS, Linux sessions and the best years in grad school. I would like to thank Dr. Nicholas Polys and Dr. Dietrich for their invaluable support throughout this process. Last but not the least, I would like to thank Sarang Joshi for his crucial help with the literature review.
# Contents

List of Figures ix

List of Tables xi

1 Introduction 1

2 Review of Literature 4

2.1 Survey: Software Radios Platforms/Frameworks 5

2.1.1 SDR Platforms 6

2.1.2 SDR Frameworks 9

2.2 Survey of performance metrics 18

2.2.1 Summary 19

3 Design and Methodology 20

3.1 System Setup 22

3.1.1 System Boundaries 23

3.1.2 Theoretical Analysis 24

3.2 System-Under-Test: CRTSv.2 27

3.3 Metrics 31
3.3.1 CPU Utilization .................................................... 31
3.3.2 Memory .............................................................. 32
3.3.3 Delays ................................................................. 33
3.3.4 Threading models .................................................. 33
3.4 Research Questions .................................................. 34

4 Results and Discussion ................................................. 35
4.1 Context Switches ...................................................... 35
4.2 CPU Utilization ....................................................... 37
4.3 Memory ................................................................. 37
4.4 Function Calls by calling object ................................. 39
4.5 Cache behaviour ...................................................... 40

5 Conclusions ............................................................... 52

6 Future Work ............................................................. 54

Appendices ................................................................. 55

Appendix A First Appendix ............................................... 56
A.1 Experiment Specifications ......................................... 56

Appendix B Massif Readings ............................................. 58
List of Figures

1.1 GNU Radio publications by subject ........................................... 3

2.1 Comparison of SDR platforms[6] .................................................. 7

2.2 SDR Platforms/Frameworks by network layer and design approach .... 8

2.3 SDR Platforms/Frameworks to be analyzed ..................................... 9

3.1 Targeted Testing Methodology ..................................................... 21

3.2 Context Switch Time with varying process size of VMWare VM ....... 23

3.3 USRP Host Interface Connectivity Options[13] .............................. 23

3.4 Memory Layer representation for testbed ...................................... 24

3.5 Network Layer representation for testbed ...................................... 25

4.1 High-level API of GR block[41] .................................................. 36

4.2 CRTSv.2 Stream graphs ............................................................ 44

4.3 Latency across tx-rx stream ....................................................... 45

4.4 OFDM Transmitter Receiver Scenario with GNU Radio and USRP .. 46

4.5 Distribution of CPU utilization w.r.t. DSP and framework libraries .. 46

4.6 CPU Utilization of Frameworks Visualized .................................... 47

4.7 Grouping framework and DSP function calls ................................. 48
4.8 Number of function calls distributed by DSP library ................. 49
4.9 Number of framework-only function calls .............................. 50
4.10 Proc statistics .............................................................. 51

B.1 Memory consumption of GNU Radio-USRP Transmitter-Receiver .... 59
B.2 CRTSv.2 Memory Consumption ............................................ 59
List of Tables

2.1 Summary of technologies extensible at the physical, MAC and network layer 10

2.2 Summary of Physical, MAC, Network layer ............................... 11

2.3 Summary of SDR frameworks ........................................... 13

3.1 Virtual Machine Specifications ........................................ 22

3.2 Summary of Latency Statistics. Lower Bound CI - Lower Bound Confidence Interval, Upper Bound CI - Upper Bound Confidence Interval ........... 29

3.3 Summary of Throughput Statistics .................................... 29

3.4 CRTS functions ............................................................ 29

3.5 Overview of GNURadio and CRTSv.2 ............................... 30

3.6 OFDM Parameters *CRC - cyclic redundancy check **FEC - Forward Error Correction ...................................................... 30

4.1 Context switches in CRTSv.2 and GNU Radio ...................... 37

4.2 Percent CPU Usage Comparison of CRTSv.2 and GNU Radio .......... 38

4.3 Summary of Memory Statistics from proc .......................... 38

4.4 Total Memory Heap Consumption Comparison of CRTSv.2 and GNU Radio 39

4.5 Memory leak comparison .................................................. 39

4.6 Virtual Memory Statistics Comparison ............................. 39
4.7  Function Calls CRTSv.2 Receiver ........................................ 40
4.8  Function Calls CRTSv.2 Transmitter .................................. 41
4.9  Function Calls of GNU Radio ............................................ 42
4.10 Last Level Cache Statistics of CRTSv.2 ............................. 42
4.11 Last Level Cache Statistics of GNU Radio ............................ 43
List of Abbreviations

CRTS v.2  Cognitive Radios Test System version 2.0

FPGA  field programmable gate array

GPP  General-purpose processors

GR  GNU Radio version 3.7.12

GUI  Graphical User Interface

HDL  Hardware Description Language

IoT  Internet-of-things

M2M  Machine-to-machine

QoS  quality-of-service

SCA  Software Communications Architecture

SDR  Software-defined Radio

Software Radio A software radio is a set of digital signal processing (DSP) primitives, a metalevel system for combining the primitives into communication functions (transmitter, channel model, receiver) and a set of target processors on which the software radio is hosted for real-time communication.[28]

TPB scheduler  thread-per-block scheduler

UHD  USRP Hardware Library
USRP  Universal Software Radio Peripheral

V2V   Vehicle-to-vehicle

WARP  Wireless Open Access Research Platform
Chapter 1

Introduction

The growing cellular market in the 1990s created a major push in the direction to use existing off-the-shelf radio front-ends with dynamically reconfigurable signal processing. Software-defined radios (SDRs) are programmable radios that provide access to all data down to the waveform, allowing programmers to implement all signal processing in software.

It is important to define the two main terms that will be used in this document; SDR platform and SDR framework. ‘An SDR platform stands for software-programmable computing equipment - a handset transceiver or network element’ [18]. SDR platforms are programmable radio frontends that provide an interfacing library to connect to a host computer. The document will refer to SDR platforms in this respect.

The following quote will be used to define SDR applications or frameworks in this document. ‘A framework such as GNU Radio enables quick reuse of algorithms, diagnostics with visualization tools, and other manipulations of signals through both simulation effects and over-the-air studies.’ [41] An SDR framework is generally the user-facing software that allows implementation, configuration, simulation etc. capabilities. Most frameworks are equipped with a user interface, either a GUI or a script.

The SDR framework under study is the waveform development and experiment management system called Cognitive Radio Test System (CR/TS). The objective of CR/TS was to provide access to the large-scale university testbed called cognitive radio network (CORN/ET) com-
prised of Universal Software Radio Peripheral (USRPs) to leverage research and education in SDR, cognitive radio, and dynamic spectrum access [48].

We evaluate this new SDR framework with reference to GNU Radio, a well-established, open-source software development toolkit that provides signal processing blocks to implement software radios. It can be used with readily-available low-cost external RF hardware to create software-defined radios, or without hardware in a simulation-like environment. It is widely used in hobbyist, academic and commercial environments to support both wireless communications research and real-world radio systems” [16]. It has been used to solve problems in applications such as space science, atmospheric radar, medicine etc. as shown in Fig. 1.1.

Chapter 2 presents a comparison analysis of features provided by various SDR frameworks. These cater to needs of reconfigurability at various layers of the network. The objective of this review was to identify patterns in design decisions for streaming real samples across software. To build better frameworks for software defined radios, it is important to perform a comparative analysis of existing software platforms/frameworks that cater to different layers of the network architecture. The aim is to provide this analysis of existing software frameworks/architectures for software defined radios. We focus on the flexibility, extensibility and configurability of these platforms/frameworks.

We use the scenario of an orthogonal-frequency digital modulation (OFDM) scheme to perform a systematic analysis that yields quantitative information about system performance and allows us to postulate origins of performance bottlenecks.

To juxtapose the SDR frameworks, a performance analysis is conducted. This provides quantitative information about the operation of CRTSv.2 and GNU Radio in terms of system performance metrics. A comparison of design decisions is provided in Chapter 3. The
objectives of this performance analysis are to determine the impact of design decisions such as threading models, buffer allocation schemes and also the usability of CRTSv.2 vs. GNU Radio.

This document evaluates quantitative system performance metrics such as CPU utilization, memory footprint and frequency of framework function calls versus frequency of signal processing function calls.
Chapter 2

Review of Literature

SDR platforms and frameworks are available in different configurations such as bottom-up FPGA implementations for a very specific wireless protocol or platform-framework pairs that allow a certain band of operation or a combination of these two. This variety leads to difficulties in selecting the right SDR platform-framework pair for simulating, testing or prototyping new protocols. We focus on layer manipulations in order to absorb new protocols. In other words, the extensibility or reconfigurability of an SDR platform or framework at a given layer of the network stack will be emphasized. This not only provides us a birds-eye view of the field as it stands but also sets the stage for a discussion of two software radios frameworks, GNU Radio and CRTSV.2. Thus, this chapter will be split into two main sections; a survey of SDR platforms/frameworks and a review of performance analyses of SDR frameworks. Two of the defining characteristics of CRTS are that it adheres to the stream-processing model adopted by SDR platforms and frameworks. It also allows reconfigurability for all layers of the network stack (caveat, physical layer reconfigurability depending on the choice of SDR platform.)

The selection of SDR platforms or frameworks depends on parameters such as band of operation, wired or wireless applications and applications for IOT, ITS (Intelligent Transport System). There is a large variety of platforms and frameworks catering to different properties of SDR’s. For example, properties such as programmability, flexibility, modularity and portability are reported in Figure 2.1. Akeela et. al. [6] provide an excellent survey of
software radios platforms in terms of design approaches whether GPP, FPGA, DSP or SoC-based and development tools. This paper provides insights into the selection of platform suitable to the application. This paper is summarized in the following table. A similar comparison will be conducted for SDR frameworks operating on these platforms. This allows insights into design approaches and design decisions. A review of SDR frameworks provided by Robert et. al. [40] discusses SDR frameworks in the context of reconfigurability. The review discusses the modularity of several frameworks and describes key components of the stream-processing model embodied by them. The stream processing model can be defined in terms of three main operations viz. gather, operate and scatter. [?] These operations are performed through filters or blocks that encapsulate functions such as fetching data from a source, performing data manipulations and sending data to a sink.

### 2.1 Survey: Software Radios Platforms/Frameworks

The objective of this section is to organize the myriad SDR platforms and frameworks into a reference for researchers. It is organized with two goals in mind, first, organization by network layer at which reconfigurability is offered. For platforms, this is generally at the physical layer but some FPGA implementations such as Atomix or Iris provided a complete system, with a customized OS that provides access to all layers. The second goal is to provide a quick reference of features advertised by the authors. With this information at hand, researchers can determine the merits and demerits of the SDR platform or framework for their applications. To that end, the first step would be to organize the obtained papers according to the TCP/IP model of network architecture and then delve into details of each. The focus is restricted to frameworks that let users program devices or implement their waveforms. Some FPGA and system-on-chip(SoC) platforms do not comply with this filter and are therefore
eliminated. Exceptions to this rule are FPGA implementations such as WARP [5] which provide physical and MAC layer extensibility targeted towards testing new platforms. We provide the tables of frameworks and platforms of interest with a brief explanation of their purpose and application. The following questions are used to collate the frameworks and organize them into layers.

1. Which network layer does a software framework cater to? Is it possible to extend or configure the waveform at this level? For example: can the physical layer be configured?

2. What protocols and implementations does this framework allow?

3. What are the limits to reconfigurability?

4. What merits do the authors claim their framework has?

5. How does the framework compare to other frameworks in its layer?

6. What are the assumptions the authors have made regarding lower or upper layers?

### 2.1.1 SDR Platforms

SDR platforms are programmable radio frontends that provide an interfacing library that connects to a host computer. Tables 2.1 shows a list of SDR platforms implemented on FPGA which are connected to a host processor. Most platforms are compatible with Linux like RHINO which runs BORPH Linux. They provide standard interfaces such as PCI or Ethernet to establish this connection to the host processor. All platforms discussed provide an extensible reconfigurable hardware interface that enables experimentation and faster processing of real-time algorithms. FPGAs in general, provide better performance
2.1. Survey: Software Radios Platforms/Frameworks

The platforms are targeted towards different use cases. For example, KUAR is targeted towards building testbeds for cognitive radios so is AirBlue. RHINO and CRUSH focus on quick prototyping with high realtime processing performance.

Most papers argue against the GPP-SDR model stating that FPGA frameworks offer higher data rates, lower latency and higher throughput at the frontend. But the complexity of the

Figure 2.1: Comparison of SDR platforms[6]
software needed is not discussed. Although most papers try to establish some workflow to make the processing easier, it generally involves knowledge of hardware description languages such as VHDL or Verilog to program FPGAs, a specific DSP library either provided by the platform or libraries such as MATLAB’s Simulink or LabView. This adds additional complexity to testing physical layer protocols.

Selecting a framework will depend on the type of protocol that needs to be run. The user will have to decide tradeoffs between design complexity and performance along with other considerations such as frequency range and extensibility along only PHYS or both PHYS and MAC layers. GNU Radio doesn’t have good performance for cross-layer whereas WARP and SORA provide high performance but difficult to modify.[34] Designers will have to decide whether their design requires cross-layer design at the lower layers and decide to trade-off coding complexity for performance.
Most platforms have two sets of controls; one PC generally running some flavour of Linux and the other an FPGA control. The way these two are provided differ from platform to platform. The higher layers such as network, transport and application are provided only in the platforms that support an OS.

### 2.1.2 SDR Frameworks

An SDR framework is generally the user-facing software that allows implementation, configuration, simulation etc. capabilities. In this section, the focus is on the frameworks that cater to most layers of the network stack. The comparison of these frameworks rests on questions as follows:

1. Which hardware devices are supported? FPGAs, DSPs or GPPs? For example: GRT or SORA do not support heterogenous hardware and support only one time of hard-
<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
<th>Extensible at layer/s</th>
<th>Software Libraries</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>KUAR [27]</td>
<td>Enable dynamic spectrum access</td>
<td>PHYS</td>
<td>KUAR Control libraries containing VHDL component library, RF control, distributed radio control and management</td>
<td>Agile transmission, distributed radio spectrum survey, channel sounding techniques</td>
</tr>
<tr>
<td>Air-Blue [34]</td>
<td>Cross-layer protocol experimentation</td>
<td>PHYS, MAC</td>
<td>Intel’s Architect’s Workbench(AWB) for FPGA/software co-design</td>
<td>802.11g transceiver, sending per-packet feedback, computing and exporting SoftPHYS hints, decoding MAC header during reception, runtime configurations through interrupts</td>
</tr>
<tr>
<td>RHINO [57]</td>
<td>Quickly prototyping radio systems for research and educational purposes</td>
<td>PHYS</td>
<td>MATLAB</td>
<td>Research and educational training and rapid prototyping</td>
</tr>
<tr>
<td>CRUSH [12]</td>
<td>Protocol design, re-configurable hardware, real-time processing</td>
<td>PHYS, MAC</td>
<td>CRUSH app using Boost library</td>
<td>High speed real time algorithms by performing signal processing close to the receiver</td>
</tr>
</tbody>
</table>

Table 2.1: Summary of technologies extensible at the physical, MAC and network layer
### Table 2.2: Summary of Physical, MAC, Network layer

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
<th>Extensible at layer/s</th>
<th>RF Frontend</th>
<th>Software</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLAWS [54]</td>
<td>Fully flexible PHY, MAC and network layer verification</td>
<td>PHYS, MAC, NETWORK</td>
<td>Zigduino transceiver</td>
<td>GNU Radio toolkit</td>
<td>FPGA PHYS transceiver, MAC processor</td>
</tr>
<tr>
<td>CrossFlow [43]</td>
<td>Cross-layer architecture platform managing wired, wireless, SDR devices using SDN</td>
<td>PHYS, MAC, NETWORK</td>
<td>USRP</td>
<td>GNU Radio toolkit</td>
<td>Physical layer adaptation, Quality of service (QoS) provisioning, Adaptive routing, Cross-layer control</td>
</tr>
<tr>
<td>IEEE 802.11b Standard-Compliant Link Layer for MATLAB-Based SDR [46]</td>
<td>Feasible framework for higher layer protocol design</td>
<td>MAC</td>
<td>USRP</td>
<td>MATLAB</td>
<td>Same node switches between transmitter and receiver, optimal selection of timing requirements, compliance with 802.11b</td>
</tr>
<tr>
<td>WARP [30]</td>
<td>Algorithmic flexibility, scalability, extensibility, capability to change according to processor updates</td>
<td>MAC</td>
<td>Maxim radio</td>
<td>WARP software</td>
<td>List of papers[5]</td>
</tr>
<tr>
<td>RcUBe[11]</td>
<td>real-time reconfigurability and optimization</td>
<td>PHYS, MAC, NETWORK</td>
<td>USRP</td>
<td>GNU Radio</td>
<td>QoS-constrained applications, cognitive radio (CR)/white-space networking</td>
</tr>
</tbody>
</table>

Table 2.2: Summary of Physical, MAC, Network layer
ware. GRT uses FPGAs where SORA relies on General Purpose Processors (GPPs). On the other hand, ALOE provides support for heterogeneous hardware devices such as FPGAs, DSPs and GPPs.

2. Are these devices reconfigurable through the framework? Hardware reconfigurability allows user to insert, delete and update hardware modules in the pipeline. For example: GRT provides a hardware module ModuleGen.

3. Which software languages are used to write SDR modules? For eg: GRT modules can be written in C++ and HDL, GNU Radio blocks can be written in C++ or Python (converted to C++ via SWIG).

4. What sort of user-interface is provided? GUI, command-line. GNU Radio provides a GUI interface to drag-and-drop blocks into a flowgraph.

5. What constraints are reported by authors?

Most of the platforms and frameworks discussed give us the following elements of streaming applications, waveform development and experiment management systems. Most frameworks encapsulate or quantize signal processing functions into a basic element. Terms encountered to describe this element are ‘block’ (GNU Radio), ‘filter’ (CRTSv.2), ‘element’ (GStreamer), ‘modules’(ALOE, DFC++), ‘component’ (REDHAWK). Elements are organized in a graph which is called ‘flowgraph’, ‘stream’, ‘pipeline’.

ALOE[18] provides a Hardware Abstraction Layer (HAL) that can communicate with the underlying hardware. Although SORA does not provide explicit support for hardware reconfigurability it uses the SIMD (Single Instruction Multiple Data) extensions in existing processors to further accelerate physical layer processing. With respect to support for building software modules, GRT provides the user with the option to create software modules
both in the user and kernel space. ALOE on the other hand, provides a set of API’s to interact software daemons in the framework.

<table>
<thead>
<tr>
<th>System</th>
<th>Supported Hardware</th>
<th>Programming Support</th>
<th>Hardware reconfigurability</th>
<th>Software reconfigurability</th>
<th>Host OS</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRTSv.2</td>
<td>GPP/DSP</td>
<td>C++ for new blocks, shell script flowgraph</td>
<td>Not tested</td>
<td>Yes</td>
<td>Linux</td>
<td>to be investigated</td>
</tr>
<tr>
<td>ALOE</td>
<td>FPGA / GPP /DSP</td>
<td>Software API</td>
<td>through Hardware Abstraction Layer (HAL)</td>
<td>Yes</td>
<td>Linux</td>
<td>resource time split into time slots which can restrict data flow</td>
</tr>
<tr>
<td>GRT</td>
<td>FPGA based</td>
<td>Software API</td>
<td>HDL</td>
<td>Yes</td>
<td>Linux</td>
<td>not reported</td>
</tr>
<tr>
<td>SORA</td>
<td>GPP</td>
<td>Software API</td>
<td>Not provided</td>
<td>Yes</td>
<td>Windows</td>
<td>Cannot guarantee the tight timing requirements of MAC layer</td>
</tr>
<tr>
<td>GNU Radio</td>
<td>FPGA / GPP</td>
<td>C++/Python for new blocks, Python or GUI for flowgraph</td>
<td>With external modules [26]</td>
<td>Yes</td>
<td>Linux, Mac, Windows</td>
<td>Slow data rates, flowgraph cannot contain loops</td>
</tr>
<tr>
<td>DFC++</td>
<td>GPP-based platforms</td>
<td>Software API</td>
<td>Not reported</td>
<td>Yes</td>
<td>Linux</td>
<td>cannot run on FPGA, thread-per-block scheduler, applications unclear</td>
</tr>
</tbody>
</table>

Table 2.3: Summary of SDR frameworks

GNU Radio[41] and REDHAWK are the most popular SDR frameworks in use currently. REDHAWK is based on the SCA specification [4]. They both offer support for a large range of software radios platforms and are therefore most versatile, hence their popularity. Additionally, they both offer plug-and-play capabilities using drag-and-drop GUIs. This
makes it very easy for anyone prototyping a system to quickly run simulations, test existing radios by connecting them via the appropriate interface to a general purpose machine. They both differ in their implementations and base languages used.

REDHAWK [1, 2, 4] is not centered on providing a specification for a framework instead it is an implementation. REDHAWK provides a data and control API, an integrated development environment (IDE), a set of component/device/service base classes, code generators, an implementation of the data and control API (ports), a Python interactive/scriptable environment, a set of reusable Components, a set of reusable Device proxies, a set of binary installation files for a set of specific operating systems (OS), and a set of debugging and diagnostic tools.

ALOE [18] provides an execution environment that provides a Hardware Abstraction Layer (HAL) to leverage the capabilities of heterogeneous multiprocessor platforms. It dynamically configuring the capabilities for deploying different waveforms at different times. Instead of CORBA it uses a message passing scheme and provides an abstract application layer via software API’s that allows interacting with the ALOE software library. ALOE currently supports GPPs, DSPs, and FPGAs based platforms.

The programming models in these different software differ greatly. GNU Radio is an event-driven framework which means that data processing is triggered by arrival or departure of samples. CRTSv.2 and REDHAWK operate on the same event-driven model. ALOE, on the other hand, works on timeslots. Processing, accumulation and evacuation of data happens at pre-determined time intervals.

In terms of threading models, GNU Radio has a thread-per-block scheduler. ALOE provides one process per module whereas in REDHAWK each component can have multiple threads. In CRTSv.2 the threading is decided at execution-time. Each filter can be assigned to the
same or different threads so filters consuming more resources can be given their own thread while fast filters can be restricted to a thread of their own.

GRT [56] is a reconfigurable SDR Platform that provides hardware reconfigurability via ModuleGen. ModuleGen uses HDL and automatically generates the corresponding interconnect and bypass logic for the hardware modules. Software modules can be created at the user and kernel level using C. GRT can be connected to a host process using a PCIe bus. GRT introduces a new Linux kernel module, shadow registers and a host-RL PCIe library in order to extract as performance from the hardware. It addresses challenges for building reconfigurable hardware blocks, with considerations for clocking domains, user/kernel cross-mode communication while maintaining modularity. GRT provides this modularity through a switching facility in which the designer can use either hardware version or software version implementations. This allows designers to incrementally improve modules. GRT provides a wireless system implementation that can be used directly on a wireless network adaptor without any modification to the code.

GNURadio [41] supports various radio frontends, can run on various operating systems and therefore used easily in field tests. GNU Radio provides accessible software and hardware which allows a simulation mode to experiment with new physical layer concepts in well-defined and repeatable scenarios [10]. The benefits of using PC implementations are that they are not limited to experiments (simulations and rapid prototyping) but can be used for modular and accessible implementation (at the cost of some latency which is recovered when using dedicated FPGA or custom platforms). [10] GNURadio does not provide good support for packet-based processing since it is stream oriented. GNURadio’s programming environment is organized around the principle of constructing a signal processing graph called a flowgraph in an integrated runtime system. The runtime system provides dynamic buffer allocation and scheduling according to fixed or dynamic I/O rates of the blocks. The
runtime system in GNU Radio is an interesting feature that we delve into when we discuss thread allocation schemes. GNURadio is not suitable for real time testbeds in the MBit range [53]. GNU Radio, by default, runs each filter on a separate thread. This is called the thread-per-block scheme.

Data Flow Control for C++ (DFC++) processing framework [44] aims to provide variable data rates, dynamic paths and flexible component design for SDR. DFC++ is useful in space and power constrained environments. DFC++ has a reference pointer-based data transport mechanisms that propagate user data between different processing components called modules. These pointers are pushed into a ring buffer by the writer module and read by the subsequent module. The ring buffer size depends on the module that pushes out the data and is a tunable parameter. This is the same transport mechanism used in CRTSv.2 making the framework lean and light. The authors have measured context-switches indirectly by measuring data throughput. The downsides of this framework are that the reported applications for the framework don’t seem to refer to the framework in particular. It uses a thread-per-block scheduling similar to GNU Radio. Also, if a particular block does not have enough input data, the thread suspends or sleeps. In future versions of CRTSv.2, threads are released if they aren’t being used which reduces waiting time. DFC++ does not work on FPGA platforms and is restricted to GPP platforms.

GStreamer [33, 35, 47] is a multimedia framework which has been used with GNU Radio’s processing framework to create SDR applications. Filters are called elements which is a C struct used to define the input, output and functions of the element. A new element can be written by encapsulating it in a plugin. This is similar to CRTSv.2 in which boilerplate is added by the opaque FilterModule class which converts the user-defined filter into a CRTSFFilter that can be included onto a stream. Connections between elements are made through pads. These pads can be thought of as ports that provide an input-output interface.
Pads need to have matching datatypes to create a link between them. Data is passed between elements (via the link) in chunks. Data transport can take place via events (control) or buffers (content). Buffers are slower because data needs to be copied into them but some specialized buffers can pass pointers. For example, the filesrc element maps a file (using mmap()) into the address space of the application and creates buffers that point to this address range[47]. This particular example illustrates the buffering scheme used in both GNU Radio and CRTSv.2 whereas, in GStreamer, this method is applied only to implement a file source element. GStreamer Events are used for control information like Stream Tags in GNU Radio. Elements need to know connection topology at compile time.

We have attempted to cover the vast area of software defined radio frameworks and platforms. We have summarized them for each layer to the best of our understanding focusing on the flexibility, extensibility and configurability of these frameworks. In selecting frameworks, there are inherent tradeoffs due to the choice of platform, whether GPP, DSP or FPGA. Akeela et. al. [6] summarize these differences in detail. Ease of use or learning curve of a technology should also be taken into consideration. Drag-and-drop GUI frameworks provide a good starting point for beginners to engage with signal processing content. Intermediate to advanced users who are comfortable with programming can reconfigure frameworks such as CRTSv.2 and GNU Radio. Real-time processing is the deciding factor in many implementations and frameworks that allow hardware reconfigurability can be chosen for these tasks. This gives the worker more control over bottlenecks introduced due to general purpose processing.
2.2 Survey of performance metrics

As seen in the previous section, there are a multitude of SDR platforms and frameworks, each specializing for a specific problem in the SDR domain whether it is fully reconfigurable MAC layer or a framework that delivers high-speed real-time signal processing. In this section, the performance metrics used in GPP-supported SDR frameworks are considered. Our test frameworks, CRTSv.2 and GNU Radio, support the USRP platform therefore, a survey of performance analysis for the subset of the SDR space namely, GPP-based SDR framework testing is merited.

In order to provide good estimates about the nature of a software system, quality-of-service (QoS) metrics need to be defined. In order to adopt a particular framework for applications such as IoT, M2M or autonomous vehicle applications, it is necessary to understand what the system requirements are. In designing such a system, the minimum processor speeds, memory, interfaces etc. need to be defined. Therefore, a software framework must provide numbers that can help system designers design requisite devices.

For example, in order to measure the memory consumption of an SCA waveform Balister et. al. [8] use the exmap program to understand the memory footprint of the application. This paper deals with the feasibility of implementing SCA waveforms in constrained-resource environments.

Software profiling provides an overview into the execution behaviour of an application specifically, the total time spent executing certain modules [24] This sort of benchmarking is useful for waveform generation frameworks as well as experiment management systems. The ALOE framework [18] evaluates processing overhead, throughput and size overhead.

GNU Radio waveforms for various SDR applications have been tested for latency (delays) [10, 36, 42, 50, 51, 53], throughput [3, 7, 10, 42, 55], packet delivery ratio [9], percentage
CPU time [7, 25] and memory usage [25], implementation complexity [53].

2.2.1 Summary

In light of this summary, we understand that most SDR frameworks follow the stream processing model of programming. Some frameworks depend on a scheduler to designate tasks whereas some have strict time-slots. Directed acyclic graphs are used to model the dataflow in these frameworks, a concept imbibed by CRTSv.2 CRTSv.2 generates a DAG for the process being run, in a fashion similar to LaTeX, where the user focuses on the building streams using a shell script and a graph is automatically generated at runtime which shows data dependencies, thread allocations and streams. Since CRTSv.2 is built in the Unix philosophy, the user can program standard Unix interfaces to tune the performance of the application. As this survey shows, some frameworks assume knowledge of hardware description languages and yet others provide their own stream processing or data flow languages. This adds to the complexity of building the system. Additionally, the closed-source nature of some projects or excessive encapsulation leads to confusion when debugging flowgraphs. CRTSv.2 can now be classed as an SDR framework operating on the stream processing model with a user-defined thread allocation system that allows access at all layers of the network stack. With the help of performance metrics used to analyze computer performance as well as ones for SDR frameworks, the next chapter lays out the details of experimental design.
Chapter 3

Design and Methodology

As reviewed in Chapter 2, software radios frameworks generally generally operate on the stream processing model. Quite a few similarities were found between software design decisions made in order to process samples in real time. In order to provide guarantees for real-time operation of SDR frameworks, it is necessary to consider delays incurred due to software processing, delays incurred due to data transfer and an analysis of the origin such delays. Theoretical analysis of delays incurred in a software radios framework have been presented below in order to compare the performance of two software frameworks running on the same testbed. We report CPU utilization, heap memory consumption, number of function calls, context switches etc. and with this information in hand we can make educated surmises about the performance of each framework.

Performance analysis as applied to experimental computer science and engineering is a combination of measurement, interpretation and communication of a computer system’s speed and capacity. In this thesis, a systematic analysis is prepared in order to consider some interactions between different components so as to provide accurate and reproducible results without disturbing the system.

In order to conduct a systematic performance analysis, the following sections describe the testbed that will be used to execute waveforms in CRTSv.2 and GNU Radio. With this demarcation of system boundaries, there follows a discussion of parameters that will provide insights into system features. These will include performance metrics such as memory
and processor utilization. We start by defining the system, writing programs to run an OFDM transmitter and receiver scenario, profiling and monitoring these programs and then, conducting a systematic statistical analysis on the numbers obtained. This is visualized in Fig.3.1.

Both frameworks were through multiple tools included in the valgrind suite along with perf and trace tools in Unix. These provided a profile of the system when it runs. The second part was to continuously monitor the system for a specified amount of time, collect system statistics from system utilities like perf, proc, top. This part is a sampling process where the bulk of the statistics were performed. The scripts to gather and publish this data were written in bash and python respectively.
3.1 System Setup

The system architecture consists of multiple USRPs connected to a host PC with an Ethernet connection, as demonstrated in Fig. 3.3. Users are granted to the radios access via virtual machines running Ubuntu 18.04.3 LTS 64-bit hosted on the host PC using VMWare ESXi hypervisor. The details are presented in the Table. 3.1. The VM was tested for context switch latency times which are shown in Fig. 3.2. This shows that a program running processes with a 16KB memory don’t fit into the cache, causing a memory read which adds latency, thus, degrading performance. The tests are conducted on a virtual machine hosted on the server and it’s important to understand in what scope the software will run. To that end, Figures 3.4 and 3.5 show the virtualization of the testbed architecture. We run the software on the same system which nullifies the effect of this architecture when comparing them. But if problems are encountered that relate to the architectural scheme itself, we know where and how to find them using this basic understanding of the VM structure.

| Hypervisor  | XCP-ng           |
| Operating System | Ubuntu 18.04.3 LTS 64-bit |
| Kernel       | Linux 4.15.0-74-generic x86_64 |
| Memory       | 15.6GiB          |
| Processor    | 2 dual-thread Intel® Xeon(R) CPU E5-2687W v4 in each physical server |
| Cores        | 24               |
| Threads      | 48               |
| Topology vCPUs | 1 socket with 16 cores per socket |
| RAM on physical server | 128GB |

Table 3.1: Virtual Machine Specifications

This architecture is used to as a testbed for both GNURadio and CRTSv.2.
3.1. System Setup

3.1.1 System Boundaries

All software frameworks will be evaluated on the same hardware and software platforms. All systems are executed on a virtual machine running Ubuntu 16.04 connected to the same set of USRP X310 software-defined radios (SDR). These USRPs are equipped with their own driver library, USRP Hardware Driver (UHD) library.
3.1.2 Theoretical Analysis

Analytical models of latency and scalability will be presented and tested against practical results.

Theoretical Analysis: Latency

Simple analytical model for OFDM transmitter modified from [42]:

\[
\Delta_{tx} = \Delta_{CRTS} + \Delta_{buffer} + \Delta_{USRP} \quad (3.1)
\]
3.1. System Setup

Figure 3.5: Network Layer representation for testbed

Simple analytical model of OFDM receiver

\[ \Delta_{rx} = \Delta_{USRP} + \Delta_{CRTS} \]  
(3.2)

Breaking it down

\[ \Delta_{CRTS} = \Delta_{stdin/stdout} + \Delta_{framing/syncfilter} \]  
(3.3)

Analytical model of circular buffer: Erlang Little's law

\[ L = \frac{\lambda - \sigma}{\mu} \]  
(3.4)

where \( L \) is length of buffer, \( \lambda \) is arrival rate, \( \sigma \) is drop-out rate and \( \mu \) is departure rate. From
This we can also say that sample rate of USRP governs how much buffers fill up.

Assuming that we do not drop any samples, we get

\[ L = \frac{\lambda}{\mu} \quad (3.5) \]

And the waiting time \( W \) is given as

\[ W = \frac{1}{\mu} \ln \frac{\lambda}{\mu} \quad (3.6) \]

The filter decides how much buffer space to allocate for pointers (pointing to processed data) being pushed into ring buffer for the next filter to consume. The filter that outputs is the writer and the one that consumes is the reader. The read and write pointer ownership to connecting buffers is with the respective reader and writer filters.

The GNU Radio runtime assigns a page of memory for the ring buffer connecting two blocks whereas in CRTSv.2 the buffer size is decided after the flowgraph is started. This value is decided based on the maximum length of data a filter requests. For a file output filter, the default is a 1024 bytes or a page in size. For the liquid framing filter used in OFDM transmission, the output buffer size is decided by Equation 3.7.

Output buffer length = (Number of subcarriers + Cyclic Prefix Length*Size of complex float) \quad (3.7)

Filter designers can choose the appropriate buffer length for the filters designed. Workers can also change the buffer length to tune performance.
3.2. System-Under-Test: CRTSv.2

**Theoretical Analysis: Scalability**

The threading scheme in CRTSv.2 makes it so that the filter designer can organize filters onto threads as opposed to the default thread-per-block assignment favoured by other frameworks. Hence, the number of threads that perform useful work can be controlled which leads to the hypothesis that lowering the number of waiting threads increases scalability of the waveform.

Thread per block model adds waiting time i.e. suspended or waiting threads do not produce useful work but system resources stay tied up nonetheless.

Processing overhead model adapted from Gomez et. al. [18]

\[
C_{CRTSv.2} = t_{\text{base}} + N t_{API} \\
T
\]

where \(T\) is the time slot duration in seconds, \(N\) is the number of application modules, \(t_{\text{base}}\) is the CPU utilization in seconds per timeslot and CRTSv.2 filter API.

3.2 System-Under-Test: CRTSv.2

The system-under-test (SUT) is the software program Cognitive Radio Test System (CRTSv.2). CRTSv.2 is a waveform development framework along with its initial objective of experiment management framework. To process the data, which is essentially a stream of samples from an software-defined radio(SDR) platform, CRTSv.2 is encapsulates signal processing functionality into filters which are then assembled onto a stream in a conveyor-belt style. This can be called a data stream processing model. Streams may be on a single or more threads. The threading scheme is described in detail in the next section. CRTS provides a command-line or web interface to control the SDR platform. The basic functions of CRTSv.2 are listed
in Table. 3.4. CRTSv.2 calls the basic processing unit as CRTSFilter or simply filter. These filters are arranged on a stream to be processed in the manner in which they are arranged and connected. This logical association between filters can be done by a shell script with appropriate options for each filter. The connections are called Channels. CRTSv.2 also provides the feature to look into filter parameters as the required software radio system is running. This lets us measure, test and validate the system.

Currently, CRTSv.2 works with the Universal Software Radio Peripheral (USRP) and therefore, it interfaces with the USRP Hardware Library (UHD) in order to interface with the software radio platform. It can interface with various signal processing libraries like liquid-dsp and GNU Radio (via pipe interface).

Thomas Rondeau [41] talks about the GNU Radio’s streaming model in the following words;

“The streaming data connections are a very natural way to represent data on the physical layer of a modem. This model typically dominates in the processing done closest to the antenna. In these situations, filtering, modulation and demodulation, synchronization and other physical-layer processes operate on the discrete sample or symbol units as they are streamed through the blocks and have no packet boundary or a relationship to bigger units of work.”[41]

CRTSv.2 works on the same design philosophy but the design decisions are distinct from GNU Radio.

Threading models in CRTSv.2 and GNU Radio differ greatly. While GNU Radio uses a thread-per-block (TPB) scheduler, in CRTSv.2 the user decides thread-to-filter assignments. The default is the single-threaded application. In CRTSv.2, the filter writer is not concerned with threading since it is a runtime decision. This is a key difference between GNU Radio and CRTSv.2. Users don’t have control over threading at runtime. Thread affinity i.e.
binding threads to a particular processor can be changed but filters/blocks cannot be placed on different threads. [37].

Another key difference is in buffers in between filters/blocks. Both use ring buffers but slightly differently. The ring buffer provides first-in-first-out (FIFO) behaviour where a single page of memory is mapped in twice and back to back.  

<table>
<thead>
<tr>
<th>Filters</th>
<th>Mean(seconds)</th>
<th>Lower Bound CI</th>
<th>Upper Bound CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0.0419</td>
<td>0.0400</td>
<td>0.0420</td>
</tr>
<tr>
<td>liquidFrame</td>
<td>0.0419</td>
<td>0.0419</td>
<td>0.0420</td>
</tr>
<tr>
<td>Transmitter</td>
<td>0.0012</td>
<td>0.0012</td>
<td>0.0012</td>
</tr>
<tr>
<td>Receiver</td>
<td>0.0050</td>
<td>0.0050</td>
<td>0.0050</td>
</tr>
<tr>
<td>liquidSync</td>
<td>0.0050</td>
<td>0.0050</td>
<td>0.0050</td>
</tr>
<tr>
<td>Output</td>
<td>0.0437</td>
<td>0.0435</td>
<td>0.0438</td>
</tr>
</tbody>
</table>

Table 3.2: Summary of Latency Statistics. Lower Bound CI - Lower Bound Confidence Interval, Upper Bound CI - Upper Bound Confidence Interval

<table>
<thead>
<tr>
<th>Filters</th>
<th>Mean(seconds)</th>
<th>Std.Deviation</th>
<th>Variance</th>
<th>LowerBoundCI</th>
<th>UpperBoundCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>liquidFrame</td>
<td>0.0416</td>
<td>0.0102</td>
<td>0.0001</td>
<td>0.0416</td>
<td>0.0416</td>
</tr>
<tr>
<td>liquidSync</td>
<td>0.0050</td>
<td>0.0025</td>
<td>0.0000</td>
<td>0.0050</td>
<td>0.0050</td>
</tr>
</tbody>
</table>

Table 3.3: Summary of Throughput Statistics

<table>
<thead>
<tr>
<th>Component</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRTSFilter or Filter</td>
<td>basic signal processing unit</td>
</tr>
<tr>
<td>FilterModule</td>
<td>software plugin created from filter</td>
</tr>
<tr>
<td>Stream</td>
<td>representation of data flow</td>
</tr>
<tr>
<td>Channel</td>
<td>connection between filters</td>
</tr>
<tr>
<td>CRTSControl</td>
<td>controls filter parameters</td>
</tr>
<tr>
<td>CRTSController or Controller</td>
<td>allows inspection of filter properties</td>
</tr>
</tbody>
</table>

Table 3.4: CRTS functions

To provide a common basis for analysis, an OFDM transmitter and receiver are implemented in both the SDR frameworks, CRTSv.2 and GNU Radio. The OFDM parameters used are

---

1Note: Page length on Unix-like systems is about 4kB whereas on Windows is 64kB
Table 3.5: Overview of GNURadio and CRTSv.2

<table>
<thead>
<tr>
<th>Parameter</th>
<th>GNURadio</th>
<th>CRTSv.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal processing</td>
<td>gr libraries such as gr::digital</td>
<td>liquid-dsp</td>
</tr>
<tr>
<td>Single signal processing unit</td>
<td>Blocks</td>
<td>Filters</td>
</tr>
<tr>
<td>Interface</td>
<td>GUI/Python</td>
<td>Command line/Web Interface</td>
</tr>
<tr>
<td>Threading</td>
<td>By default, each filter on a separate thread</td>
<td>User decides which filter/s on which thread</td>
</tr>
<tr>
<td></td>
<td>Blocks connect in a flowgraph</td>
<td>Filters connected in a graph via channels</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OFDM Parameter</th>
<th>CRTSv.2</th>
<th>GNU Radio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Center frequency</td>
<td>915.5 MHz</td>
<td>915.5 MHz</td>
</tr>
<tr>
<td>Data rates</td>
<td>400k</td>
<td>400k</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>10MHz</td>
<td>10MHz</td>
</tr>
<tr>
<td>Modulation Type</td>
<td>QAM</td>
<td>QAM</td>
</tr>
<tr>
<td>Number of subcarriers</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Length of cyclic prefix</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Payload length</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td>CRC*</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>FEC**</td>
<td>128</td>
<td>128</td>
</tr>
</tbody>
</table>

Table 3.6: OFDM Parameters

*CRC - cyclic redundancy check
**FEC - Forward Error Correction

shown in Table 3.6. We have accounted for the variation in data rates when we obtain the relevant data. The variation in number of bits would result in changes to the storage and processing time. The effects of the varying data can be seen in the measures of CPU utilization, context switches and number of function calls. Essentially, we have removed the effects of UHD and USRP in terms of transmission, reception, transport to higher layers thus, isolating the software frameworks from the platform they depend on. So we can say that the numbers we report in terms of software processing in the framework (not including processing at the SDR-level) are comparable.

Despite the difference in payload lengths, the experiment was checked for differing payload length values as well but the results showed various similarities. The function call numbers, CPU utilization, memory footprint and context switch numbers are quite close to the ones for matching payloads. This leads us to conclude that payload length does not make much
of a difference on the metrics.

3.3 Metrics

3.3.1 CPU Utilization

The OFDM transmitter and receiver are executed at different bit rates, and their CPU consumption measured for a duration of one second. [15] This will give us information about overall CPU consumed by the same block/system in two different frameworks.

We use perf and flamegraphs for stack traces, valgrind [32] with massif for memory usage, valgrind with callgrind for call graphs, valgrind with memcheck for memory leaks in order to profile, benchmark and compare software frameworks GNU Radio and CRTSv.2

Cost attributes given in KCachegrind are related to a particular event types and depend on the tool being used. If the tool being used is Cachegrind, then the cachegrind.out.<pid> file will contain the fields such as LL Data Write Miss, LL Data Read Miss etc. For the tool callgrind, the fields would be Instruction Fetch and Cycle Estimation. Standard code structure contains functions that call each other therefore, the KCachegrind tool distinguishes between the cost of the function itself an the cost of the function as well as all the functions that it calls. The cost of the function alone is called Self Cost and the cost of the function call trace is called Inclusive costs. 'Self' is sometimes also referred to as 'Exclusive' costs. This can be illustrated by looking at the main() function which is the entry-point to C/C++ code. The Self cost of main is negligible since most of the work is being done in another function. But the Inclusive cost of main() is 100% since all the functions are being called from it[22]. We will make use of Self cost wherever possible to make conclusions. If Inclusive Cost is used, we justify the use of Inclusive Cost.
Cachegrind[31], as mentioned above, outputs a cachegrind.out.<pid> file. It is an inbuilt tool with valgrind that simulates a cache hierarchy and conducts branch prediction, if necessary. It simulates a machine with independent first-level instruction and data caches (I1 and D1) and a unified second-level cache (L2) which matches the configuration of many modern machines. But for our testbed setup this assumption is not valid. KCachegrind detects this difference and then simulates only the first and the last-level caches. The reasons for this choice are the first level caches have low associativity and it’s important to understand how the code interacts with these caches. The last level cache interaction affects the runtime i.e. more cache misses at this level will lead to high wait times. Since latency is an important measure for stream processing applications, we focus on the last level read and write misses to estimate how many cycles were lost trying to retrieve data from main memory. This is of highest relevance in the case of context switches more specifically, nonvoluntary context switches. This is explained in more detail in the following subsection on Delays 3.3.3.

### 3.3.2 Memory

In addition to memory leak checks, valgrind’s default tool, memcheck, helps analyze the memory footprint of a system. Memory leaks need to be reduced in order to improve memory utilization. Apart from this memory usage, it is important to see heap memory consumption so that we can understand the real-time execution of the program. In order to obtain costs for memory, the source files or ELF objects would have to be classified as framework, signal processing or UHD specific memory operations. For the other cases like function calls and cache behaviour this tracing is possible through kcachegrind, which visualizes the profile data. But this is not the case for memory because the same memory-related files such as dl-machine, strdup.c are used in any of the aforementioned contexts. Hence, we limit the scope of our measurements to the total memory consumption and hope to amortize the effects of
3.3. Metrics

UHD signal processing by sampling the data such that we obtain virtual memory size mean values that are statistically significant.

3.3.3 Delays

Universal Software Radio Peripherals (USRPs) are commonly used hardware platforms for software radios testbeds. USRPs, devised by Ettus Research, comes with USRP Hardware Driver libraries that act as an interface between the host computer and the hardware. The latency incurred while running any radio application can therefore, be broken down into three key areas of interest; software latency incurred due to complexity of signal processing tasks, bus latency while communicating between host computer and hardware platform and hardware latency due to buffering of samples in the USRP \[39, 42, 49, 53\]. A process might have to switch between kernel mode and user mode to handle syscalls. This causes the process cache to either flush or replace items with the ones it needs to execute in kernel mode. When the process switches back to user mode, the data that it needs might not be in the cache causing it to reach back through the memory hierarchy. Context switches in a system causes unnecessary delays. Involuntary context switches cause more lost cycles-per-instructions than voluntary context switches \[29\].

3.3.4 Threading models

Inter- vs. intra-block parallelism. Some I/O operations like input from USRP and displaying spectrum can happen at the same time but operations with data dependencies like signal processing blocks sending data to graphing tool may not be parallelizable. To investigate the effects of parallelization on SDR performance, we measure the number of samples at various intervals in a given flowgraph or system. \[19\]
3.4 Research Questions

The following research questions will be addressed in this document:

- Does a lighter framework such as CRTSv.2 perform faster than a comparatively feature-heavy GNU Radio? A comparison between the two frameworks can provide certain use cases that perform better in either case. This helps researchers select the framework that works best for their needs.

- CRTSv.2 provides a flexible thread allocation scheme in which the system designer can assign blocks or sets of blocks to threads. To test the impact of this feature, we compare various threading schemes with each other and find the optimum thread allocation scheme.

- How does CRTS manage the tradeoffs between ease of use and speed of execution. The initial purpose of CRTS was as an experiment management framework but it evolved into an SDR waveform development framework. This broad question can be answered with inferences drawn from investigation into the above questions.
Chapter 4

Results and Discussion

In this section, we discuss the CPU utilization, memory usage, number of function calls and number of context switches of CRTSv.2 and GNU Radio v.3.7.12. These metrics will help decide which software framework performs relatively better. CRTSv.2 compiles stream graphs as shown in Fig.4.2 and GNU Radio flowgraph is shown in Fig.4.4. The test case is used OFDM transmitters and receivers. GNU Radio uses its own inbuilt digital signal processing library while CRTSv.2 uses liquid-dsp [14]. Fine-grained latency measures could not be taken for GNU Radio but are reported for CRTSv.2 with 95% confidence as shown in Table.??.

4.1 Context Switches

According to the manpage for pidstat[17], ‘a voluntary context switch occurs when a task blocks because it requires a resource that is unavailable’ and ‘a involuntary context switch takes place when a task executes for the duration of its time slice and then is forced to relinquish the processor.’

For larger processes, the caches get polluted leading to larger context switching times. Caches get more polluted as the number of processes increases. The context switch time is composed of the switch time, the time it takes to restore all of the process state and cache state and the time for the cache misses on larger processes. All these factors contribute to increasing
### Constructor
- Inherit from a gr::block
- Setup Input and Output signatures
- Setup message input and output ports
- assign input message handler functions
- Set advanced scheduler interactions
- Any initialization required for your DSP code

### Message Handlers
- Receiver a GNU Radio message
- Return nothing
- Bound to a message input in constructor

```cpp
void general_work()
{
    // Receivers
    // Input and output data stream connections
    // Amount of data available on the input streams
    // Amount of space available on output stream

    // Access tag stream to get or insert tags
    // Output messages on output ports

    // Your DSP code goes here

    // Tell scheduler how much block has produced and consumed
}
```

**Figure 2.8** The high-level API for programming a GNU Radio block.

**Figure 4.1:** High-level API of GR block[41]

context switch times as the process size increases [45]. More involuntary context switches can mean that a large portion of time is spent in refilling caches which degrades system performance. GNU Radio shows more involuntary context switches than CRTSv.2 which means
that it may perform slower than CRTSv.2 unless large registers and caches are provided to amortize the effect of a context switch. The numbers are presented in Table 4.1.

<table>
<thead>
<tr>
<th></th>
<th>CRTSv.2</th>
<th>GNU Radio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Voluntary context switches</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tx</td>
<td>2335</td>
<td>2383</td>
</tr>
<tr>
<td>Rx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>4718</td>
<td>14</td>
</tr>
<tr>
<td>Involuntary context switches</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.1: Context switches in CRTSv.2 and GNU Radio

### 4.2 CPU Utilization

The CPU utilization numbers show that CRTSv.2 has a minimal impact on CPU usage and the focus remains on signal processing. This shows that the streaming application itself provides a good lightweight framework for SDR applications without unnecessary software overheads. Running a Student test or a t-test gives a p-value less than 10%. This means we can say that the average CPU utilization for both frameworks is not incidental and we can say with certainty that the differences observed are statistically significant. The null hypothesis is that both frameworks have the same CPU utilization. However, the p-values indicate that the null hypothesis has to be rejected which means that the two frameworks have different mean values and we can say with certainty that the difference is not due to random effects. A visualization is presented in Fig. 4.6. A distribution of the CPU utilization with respect to framework and DSP is shown in Fig. 4.5.

### 4.3 Memory

We compare the total amount of virtual memory, percentage of memory, resident memory and shared memory used by the GNU Radio and CRTSv.2. This data was collected from the
%CPU usage stats

<table>
<thead>
<tr>
<th></th>
<th>CRTSv.2</th>
<th>GNU Radio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>Tx</td>
<td>Rx</td>
</tr>
<tr>
<td></td>
<td>7.5</td>
<td>2.9</td>
</tr>
<tr>
<td>Std. dev.</td>
<td>0.4</td>
<td>0.3</td>
</tr>
<tr>
<td>Variance</td>
<td>0.2</td>
<td>0.09</td>
</tr>
<tr>
<td>Lower bound CI</td>
<td>7.4</td>
<td>2.9</td>
</tr>
<tr>
<td>Upper bound CI</td>
<td>7.6</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Table 4.2: Percent CPU Usage Comparison of CRTSv.2 and GNU Radio

Linux utility called top. The manual page of the top utility define the terms virtual memory, shared memory size and resident memory as follows; ‘Virtual memory includes all code, data and shared libraries plus pages that have been swapped out and pages that have been mapped but not used. Resident memory Size (KiB) is a subset of the virtual address space (VIRT) representing the non-swapped physical memory a task is currently using. Shared Memory Size (KiB) is a subset of resident memory (RES) that may be used by other processes. It will include shared anonymous pages and shared file-backed pages. It also includes private pages mapped to files representing program images and shared libraries’[38].

Data is also collected from the Linux utility proc which provides more detailed information about the process. The proc utility tells us the peak virtual memory size, the size of data, size of stack, number of voluntary/nonvoluntary switches as well as number of threads spawned by the process.

<table>
<thead>
<tr>
<th>Fields</th>
<th>CRTSv.2</th>
<th>GNU Radio</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIRT</td>
<td>799764 KiB</td>
<td>7781736 KiB</td>
<td>p&lt;0.001</td>
</tr>
<tr>
<td>RES</td>
<td>45896 KiB</td>
<td>246860 KiB</td>
<td>p&lt;0.001</td>
</tr>
<tr>
<td>SHR</td>
<td>37880.0 KiB</td>
<td>114104 KiB</td>
<td>p&lt;0.001</td>
</tr>
<tr>
<td>MEM%</td>
<td>0.2</td>
<td>0.79</td>
<td>p&lt;0.001</td>
</tr>
</tbody>
</table>

Table 4.3: Summary of Memory Statistics from proc

With these memory statistics, it is reasonable to conclude that GNU Radio consumes a significant amount of process memory.
4.4. Function Calls by calling object

Valgrind’s heap profiler takes 85 snapshots of heap memory utilization to provide the numbers shown in Table 4.4.

<table>
<thead>
<tr>
<th>Total Memory Heap Consumption</th>
<th>CRTSv.2</th>
<th>GNU Radio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tx 2.9MiB</td>
<td>Rx 2.9MiB</td>
<td>5.8MiB</td>
</tr>
</tbody>
</table>

Table 4.4: Total Memory Heap Consumption Comparison of CRTSv.2 and GNU Radio

Memory leak comparison is performed using valgrind’s memcheck. The results are summarized in Table 4.5.

<table>
<thead>
<tr>
<th>Event</th>
<th>CRTSv.2</th>
<th>GNU Radio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes definitely lost</td>
<td>8570</td>
<td>471(lost in new)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1632(lost in malloc)</td>
</tr>
</tbody>
</table>

Table 4.5: Memory leak comparison

<table>
<thead>
<tr>
<th>Event</th>
<th>CRTSv.2</th>
<th>GNURadio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak Virtual Memory</td>
<td>449108.0</td>
<td>6568481.666</td>
</tr>
<tr>
<td>Size of Virtual Memory Data</td>
<td>78252.857</td>
<td>771406.333</td>
</tr>
<tr>
<td>Size of Virtual Memory Stack</td>
<td>138.857</td>
<td>132.000</td>
</tr>
<tr>
<td>Number of threads</td>
<td>10.000</td>
<td>80.000</td>
</tr>
<tr>
<td>Number of voluntary context switches</td>
<td>2062.143</td>
<td>848.000</td>
</tr>
<tr>
<td>Number of nonvoluntary context switches</td>
<td>31.214</td>
<td>476.250</td>
</tr>
</tbody>
</table>

Table 4.6: Virtual Memory Statistics Comparison

4.4 Function Calls by calling object

As shown in Tables 4.7, 4.8 and 4.9, function calls are split by the object making the most calls. In the case of CRTSv.2 a large portion of the function calls are attributed to the underlying liquid-dsp library that performs the digital signal processing whereas in GNU Radio, most of the function calls are due to the Python wrapper which translates user
Table 4.7: Function Calls CRTSv.2 Receiver

<table>
<thead>
<tr>
<th>Executable file</th>
<th>Self Cost (Number of Function Calls)</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>libliquid.so</td>
<td>55.25</td>
<td>Highest contribution to total cost by signal processing library; OFDM filters _mulsc3 costs 7.92 for complex arithmetic in liquid dsp</td>
</tr>
<tr>
<td>libuhd.so.3.14</td>
<td>02.28</td>
<td>UHD allocations</td>
</tr>
<tr>
<td>libc-2.27.so</td>
<td>10.79</td>
<td>free, malloc, memcpy</td>
</tr>
<tr>
<td>libm-2.27.so</td>
<td>6.50</td>
<td>LIBM standard C library of basic mathematical functions[23]</td>
</tr>
<tr>
<td>lib-boost_regex.so.1.65.1</td>
<td>5.18</td>
<td>called by uhd::rfnoc not CRTSv.2</td>
</tr>
<tr>
<td>libstdc++.so.6.0</td>
<td>4.31</td>
<td>string operations etc.</td>
</tr>
<tr>
<td>rx.so</td>
<td>0.04</td>
<td>Filter rx</td>
</tr>
<tr>
<td>TOTAL</td>
<td>94.31</td>
<td></td>
</tr>
</tbody>
</table>

specifications into C++. GNU Radio’s signal processing on the other had takes less time maybe due to the use of optimized kernel libraries such as VOLK. Function calls of the GNU Radio framework are much more than those for the CRTSv.2 framework which leads to the conclusion that CRTSv.2 is a lightweight framework. This is visualized in Figures 4.8 and 4.9. To illustrate the lightweight nature of the CRTS framework, Fig.4.7 shows an approximate distribution of the function calls with respect to framework and digital signal processing respectively.

4.5 Cache behaviour

Last-level cache has the most influence on runtime, as it masks accesses to main memory. Therefore, read and write misses to last-level cache will be considered when performance
4.5. Cache behaviour

<table>
<thead>
<tr>
<th>CRTSv.2 Tx</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF Object</td>
<td>Self Cost</td>
<td>Significance</td>
</tr>
<tr>
<td>libliquid.so</td>
<td>44.06</td>
<td>Highest contribution to total cost by signal processing library (liquid repack bytes:9.54, ofdmflexframegen:8.42)</td>
</tr>
<tr>
<td>liquid-Frame.so</td>
<td>17.49</td>
<td>std::complex&lt;float&gt;::operator*= (float)</td>
</tr>
<tr>
<td>libc-2.27.so</td>
<td>12.64</td>
<td>memory operations (print, free, malloc, memmove) 7</td>
</tr>
<tr>
<td>libfftw3f.so.-3.5.7</td>
<td>8.77</td>
<td>FFTW C subroutine library for computing the discrete Fourier transform (DFT) [20]</td>
</tr>
<tr>
<td>libuhd.so.-3.14.0</td>
<td>8.37</td>
<td>UHD</td>
</tr>
<tr>
<td>crts_radio</td>
<td>4</td>
<td>CRTSv.2 FilterModule, CRTSFilter, CRTSControl, Stream, RingBuffer, input/output functions</td>
</tr>
<tr>
<td>logger.so</td>
<td>2.33</td>
<td>get number of items written and read for each filter</td>
</tr>
<tr>
<td>tx.so</td>
<td>0.13</td>
<td>Filter tx</td>
</tr>
<tr>
<td>TOTAL</td>
<td>97.79</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.8: Function Calls CRTSv.2 Transmitter

benchmarking the two applications. A large portion of cache read misses are due to GNU Radio’s framework (77%) whereas CRTSv.2 functions do not contribute as much to the read misses in the last level cache. These are illustrated in Tables 4.10 and 4.11.
GNU Radio

<table>
<thead>
<tr>
<th>ELF Object</th>
<th>Self Cost</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>python2.7</td>
<td>19.94</td>
<td>Python wrapper functions contribute to the total cost</td>
</tr>
<tr>
<td>libc.2.27.so</td>
<td>18.82</td>
<td>~13 in free, malloc, getc, memmove, memset etc. (Memory Operations)</td>
</tr>
<tr>
<td>libcuhd.so.3.14.0</td>
<td>13.88</td>
<td>UHD allocations</td>
</tr>
<tr>
<td>libgnuradio-runtime-3.7.12.0.so.0.0.0</td>
<td>13.33</td>
<td>GNU Radio Runtime operations such as scheduling</td>
</tr>
<tr>
<td>libgnuradio-blocks-3.7.12.0.so.0.0.0</td>
<td>8.85</td>
<td>Basic blocks in GR</td>
</tr>
<tr>
<td>libgnuradio-pmt-3.7.12.0.so.0.0.0</td>
<td>5.58</td>
<td>Polymorphic type classes used to convert between datatypes</td>
</tr>
<tr>
<td>libgnuradio-digital-3.7.12.0.so.0.0.0</td>
<td>4.38</td>
<td>Signal processing library; OFDM specific blocks</td>
</tr>
<tr>
<td>libvolk.so.1.4</td>
<td>3.96</td>
<td>VOLK Vector-Optimized Library of Kernels, SIMD-specific handwritten math functions [16]</td>
</tr>
<tr>
<td>libstdc++.so.6.0.25</td>
<td>3.41</td>
<td>operator new, string etc</td>
</tr>
<tr>
<td>ld-2.27.so</td>
<td>2.19</td>
<td>linker functions such as lookups, open, mmap, memcpy</td>
</tr>
<tr>
<td>TOTAL</td>
<td>88.74</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.9: Function Calls of GNU Radio

CRTSv.2 Transmitter Last-level Cache Events

<table>
<thead>
<tr>
<th>Source File</th>
<th>LL Data Read Miss</th>
<th>LL Data Write Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>dl-machine.h</td>
<td>44.68</td>
<td>17.29</td>
</tr>
<tr>
<td>dl-lookup.c</td>
<td>25.12</td>
<td>0.20</td>
</tr>
<tr>
<td>do-rel.h</td>
<td>14.24</td>
<td>0</td>
</tr>
</tbody>
</table>

CRTSv.2 Receiver Last-level Cache Events

<table>
<thead>
<tr>
<th>Source File</th>
<th>LL Data Read Miss</th>
<th>LL Data Write Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>dl-machine.h</td>
<td>44.78</td>
<td>17.18</td>
</tr>
<tr>
<td>dl-lookup.c</td>
<td>25.31</td>
<td>0.23</td>
</tr>
<tr>
<td>do-rel.h</td>
<td>14.25</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.10: Last Level Cache Statistics of CRTSv.2
<table>
<thead>
<tr>
<th>Source File</th>
<th>LL Data Read Miss</th>
<th>LL Data Write Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>python wrappers</td>
<td>73.20</td>
<td>57.99</td>
</tr>
<tr>
<td>dl-machine.h</td>
<td>8.95</td>
<td>5.17</td>
</tr>
<tr>
<td>dl-lookup.c</td>
<td>3.28</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.11: Last Level Cache Statistics of GNU Radio
(a) OFDM transmitter running three filters on the same thread
(b) OFDM receiver running three filters on the same thread

Figure 4.2: CRTSv.2 Stream graphs
4.5. Cache behaviour

Figure 4.3: Latency across tx-rx stream
Figure 4.4: OFDM Transmitter Receiver Scenario with GNU Radio and USRP

Figure 4.5: Distribution of CPU utilization w.r.t. DSP and framework libraries
Figure 4.6: CPU Utilization of Frameworks Visualized
Figure 4.7: Grouping framework and DSP function calls
Comparison of DSP libraries

System monitor: valgrind with callgrind

- liquid-dsp framing filter
- liquid-dsp sync filter
- GNU Radio inbuilt gr::digital

Figure 4.8: Number of function calls distributed by DSP library
Comparison of framework overhead

System monitor: valgrind with callgrind

- CRTS framework transmitter: 4%
- CRTS framework receiver: 6%
- GNU Radio framework: 48%

Figure 4.9: Number of framework-only function calls
Figure 4.10: Proc statistics
Chapter 5

Conclusions

To a programmer familiar with the basics tenets of the Unix philosophy and a grasp of shell scripting, CRTSv.2 provides a minimal, lightweight framework to perform experiments on the CORNET testbed. The lightweight properties of the framework have been shown by its CPU utilization which is considerably lower than that of GNU Radio. It is also lean in terms of memory usage, using approximately 3MB of heap to conduct its operations. This compared to the heavy 25MB that GNU Radio reports, we can see how CRTSv.2 will be useful in resource-constrained environments. On the other hand, most of the memory in CRTSv.2 is being consumed by the underlying signal processing library, liquid-dsp whereas the cost of signal processing in GNU Radio has been optimized with the help of VOLK libraries and is distinctly lower. After conducting a systematic performance analysis of two SDR frameworks, CRTSv.2 shows the characteristics of an optimized, lightweight framework with a low memory footprint. CRTSv.2 does not have a high learning curve for users familiar with SDR frameworks. The web interface that goes with it makes it even easier to run simulations. Flowgraph writers need to understand basic shell scripting whereas filter writers would need to understand how the start, stop and input functions work in order to build custom filters. This complexity in filter design remains a constant throughout the SDR frameworks studied due to the nature of the field. To address concerns of building new flowgraphs, the worker need only refer to the example scenarios to be able to write their own flowgraphs with custom thread allocation schemes that can optimize the performance of real-time signal processing.
Due to this performance analysis, we can definitively state that CRTSv.2 is a competent SDR framework for simulation and implementation.
Chapter 6

Future Work

The newest version of CRTS will be called Quickstream. Based on the performance tests, a modular streaming application that can integrate with the visualization capabilities of CRTS would help specific development. The threading model will be optimized to reduce wait times for threads. In order to evaluate the flexibility of the CRTSv.2 framework, it should be tested with different SDR platforms as well as optimized signal processing libraries. Due to the transparency of the coding structure, it is possible to optimize the framework for specific applications and integrating CRTSv.2 in radar applications etc. can prove its use beyond that of an experiment management system. A better testing system can be employed for GNU Radio to obtain metrics without significantly disturbing the system.
Appendices
Appendix A

First Appendix

A.1 Experiment Specifications

CRTSv.2 and USRP flowgraph for OFDM transmitter and receiver scenario.

```
#!/bin/bash
set -e
cd $(dirname ${BASH_SOURCE[0]})
source usrp_config
crts_radio="../bin/crts_radio"

#.termRun

#.termRun uhd_fft -f 915.0e6 -s 10.5e6 --args $USRP3

./termRun "cat /dev/urandom |\n$crt$radio\"`
A.1. Experiment Specifications

-f stdin
-f liquidFrame
-f tx [ --uhd $USRP1 --freq 915.5 --rate 0.4 --gain 15 ]
-C logger [ --file stdinLogs/$fname stdin totalBytesOut 
--file frameLogs/$fname liquidFrame totalBytesIn totalBytesOut 
--file txLogs/$fname tx totalBytesIn ]
-D"

# 915.5 MHz receiver
./termRun "$crts_radio"
-f rx [ --uhd $USRP2 --freq 915.5 --rate 0.4 --gain 0 ]
-f liquidSync
-f stdout
-C logger [ --file rxLogs/$fname rx freq totalBytesOut 
--file syncLogs/$fname liquidSync totalBytesIn totalBytesOut 
--file stdoutLogs/$fname stdout totalBytesIn ]
-D "
hexdump -v"
Appendix B

Massif Readings
Figure B.1: Memory consumption of GNU Radio-USRP Transmitter-Receiver

(a) Memory consumption of CRTSv.2-USRP (b) Memory consumption of CRTSv.2-USRP Receiver

Figure B.2: CRTSv.2 Memory Consumption
Bibliography


[54] Bertold Van den Bergh, Tom Vermeulen, Marian Verhelst, and Sofie Pollin. Claws: Cross-layer adaptable wireless system enabling full cross-layer experimentation on real-

