ECE496: Final Report

Integrating Wireshark with an FPGA BPF Engine (FFShark) for Packet Analysis of High Speed Networks (100G)

Project Number: 2020174

Supervisor Name: Professor Paul Chow

Administrator Name: Inci McGreal (Section 5)

Department: The Edward S. Rogers Sr. Department of Electrical and Computer Engineering

Students: Kanver Bhandal (kanver.bhandal@mail.utoronto.ca), Alexander Buck (alexander.buck@mail.utoronto.ca), Tobias Rozario (tobias.rozario@mail.utoronto.ca)

Date of submission: March 30, 2021

Group Final Report Attribution Table

This table should be filled out to accurately reflect who contributed to each section of the report and what they contributed. Provide a **column** for each student, a **row** for each major section of the report, and the appropriate codes (e.g. 'RD, MR') in each of the necessary **cells** in the table. You may expand the table, inserting rows as needed, but you should not require more than two pages. The original completed and signed form must be included in the hardcopies of the final report. Please make a copy of it for your own reference.

Section	Student Names		
	Kanver	**Alex**	**Tobias**
Group Highlights	RD	ET	
Acknowledgments		RD	
Executive Summary			RD
1.0		RD	
1.1	RS	RS, RD	RS
1.2		ET	RD
2.0			RD
2.1	RD		
2.2			RD
2.3		RD	
2.4	RS	RS, RD	RS
2.5	RD, MR	ET	
3.0		RD	
3.1		RD	
3.1.1	RS, RD		
3.1.2		RS	RS, RD
3.2		RS, MR	RD
4.0	RD, MR	ET	
5.0	MR		RD
Appendix A	RD		
Appendix B		RD	
Appendix C			RD
Appendix D		RD	
All	ET, FP	ET, FP	ET, FP

Abbreviation Codes:

Fill in abbreviations for roles for each of the required content elements. You do not have to fill in every cell. The "**All**" row refers to the complete report and should indicate who was responsible for the final compilation and final read through of the completed document.

RS – responsible for research of information

RD – wrote the first draft
MR – responsible for major revision
ET – edited for grammar, spelling, and expression
OR – other
“All” row abbreviations:
FP – final read through of complete document for flow and consistency
CM – responsible for compiling the elements into the complete document
OR - other
If you put OR (other) in a cell please put it in as OR1, OR2, etc. Explain briefly below the role referred to:
OR1: enter brief description here
OR2: enter brief description here

Signatures

By signing below, you verify that you have read the attribution table and agree that it accurately reflects your contribution to this document.

Name	Kanver Bhandal	**Signature**	Kanver Bhandal	**Date:**	Mar. 30, 2021
Name	Alexander Buck	**Signature**	Alexander Buck	**Date:**	Mar. 30, 2021
Name	Tobias Rozario	**Signature**	Tobias Rozario	**Date:**	Mar. 30, 2021

Voluntary Document Release Consent Form[1]

To all ECE496 students:

To better help future students, we would like to provide examples that are drawn from excerpts of past student reports. The examples will be used to illustrate general communication principles as well as how the document guidelines can be applied to a variety of categories of design projects (e.g. electronics, computer, software, networking, research).

Any material chosen for the examples will be altered so that all names are removed. In addition, where possible, much of the technical details will also be removed so that the structure or presentation style are highlighted rather than the original technical content. These examples will be made available to students on the course website, and in general may be accessible by the public. The original reports will not be released but will be accessible only to the course instructors and administrative staff.

Participation is completely voluntary and students may refuse to participate or may withdraw their permission at any time. Reports will only be used with the signed consent of all team members. Participating will have no influence on the grading of your work and there is no penalty for not taking part.

If your group agrees to take part, please have all members sign the bottom of this form. The original completed and signed form should be included in the hardcopies of the final report.

Sincerely,
Khoman Phang
Phil Anderson
ECE496Y Course Coordinators

Consent Statement

We verify that we have read the above letter and are giving permission for the ECE496 course coordinator to use our reports as outlined above.

Team #: 174 Project Title: Integrating Wireshark with an FPGA BPF Engine (FFShark) for Packet Analysis of High Speed Networks (100G)

[1] This form will be detached from the hardcopy of the final report. Please make sure you have nothing printed on the back page.

Supervisor: Prof. Paul Chow

Administrator: Inci McGreal

Name	Kanver Bhandal	**Signature**	Kanver Bhandal	**Date:**	Mar. 30, 2021
Name	Alexander Buck	**Signature**	Alexander Buck	**Date:**	Mar. 30, 2021
Name	Tobias Rozario	**Signature**	Tobias Rozario	**Date:**	Mar. 30, 2021

Group Highlights (Author: Kanver Bhandal)

Through the past two semesters we have been working on integrating a GUI, Wireshark, into FFShark and now have accomplished our goal. By connecting a GUI to FFShark, we increased the usability of FFShark as no programming knowledge is required to read filtered packets from FFShark. We thoroughly tested our design to make sure that there are no unexpected bugs and have documented the limitations of our design for our users. Additionally, we measured the performance of our design and found that it has reached 59 Mbps using a C implementation of our code with no locking and several optimizations.

To reach this point we faced many challenges. Some were due to hardware bugs in the FFShark design such as FFShark being limited to packets that are a multiple of 8 bytes long. Others were inside the software library sshdump which prevents programs from being terminated until they print onto the terminal. These bugs were discovered during our testing and verification and we either developed a fix for the bug or a workaround. If a bug could not be fixed we documented how and why it can occur so that any user of our project would be aware of it.

Kanver was responsible for creating the hardware interface to FFShark. He wrote the code on how to interact with the hardware that sends and reads packets from FFShark. Additionally, he performed testing and verification on the hardware interface and debugged any issues that were related to the hardware though hardware simulation or probes connected to the hardware.

Alex was responsible for setting up the connection between the ARM chip connected to FFShark and Wireshark and creating a script that could generate random packets. He also wrote the PCAP formatting script so that packets read from FFShark could be sent to Wireshark in PCAP format. Finally, he performed overall verification of our design, throughput measurements, and optimizations.

Tobias was responsible for writing the FFShark driver which read out packets from FFShark and for creating a script that randomly sends packets into FFShark. Additionally, he conducted component level testing for the different modules of the driver. He also added profiling code in the FFShark driver to find bottlenecks and rewrote our code in C to improve performance.

Individual Contributions (co-authors: Kanver Bhandal, Alexander Buck, Tobias Rozario)

Contributions of Kanver Bhandal

The following table summarizes all of Kanver's contributions throughout the project.

Contribution	**Description**
ARM communication with FFShark research	Researched how the ARM chip can read out the filtered packets from FFShark and how the ARM chip can send in packets. Found two main options: the AXI Stream FIFO and AXI Stream DMA.
Receive FIFO to store output of FFShark	Added a Receive FIFO to the hardware interface and connected it to FFShark.
ARM computer to FFShark Communication	Wrote code on how to send packets to the Send FIFO and how to read packets from the Receive FIFO from the ARM chip.
Verify sendfile script	Verified that a packet could be sent into FFShark using the sendfile script and be read out from the Receive FIFO and printed to the terminal.
Integration and end to end single packet testing	Wrote code to send a random packet into FFShark and then read out the packet. Then write the packet into PCAP format and send it to Wireshark. The packet should be displayed on the Wireshark GUI. Tested that a single packet could be sent.
Code rewrite to use new drivers	Modified FFShark driver to use the new *libmpsoc.py* library for reading and writing to AXI interfaces from the ARM chip.
FFShark packet size bug	Investigated the root cause of the multiple of 8 bytes bug and started working on a potential fix. Still need to verify the fix under simulation and then in real hardware.
Root cause of dropped packets when first starting FFShark Driver	Investigated and root caused why packets were being dropped when first starting the FFShark Driver.
Investigating improving AXI Stream FIFO performance	Configured the AXI Stream FIFOs to use the AXI Full interface instead of AXI Lite and to use cut through mode instead of store and forward mode. Unfortunately did not improve the performance of our design so the modifications were not kept.

Contributions of Alexander Buck

The following table summarizes all of Alexander's contributions throughout the project.

Contribution	**Description**
Extcap research	Researched existing plugin utilities in Wireshark for custom capture interfaces.
Sshdump research	Researched and recommended using *sshdump* as our capture interface.
Sshdump to ARM computer connection	Created initial connection from Wireshark to ARM chip.
Random Packet Generator	Created a generator for our test packets.
Locking Mechanism	Initial implementation of locking mechanism for common bus.
Send Random Packets to Wireshark	Simulated the hardware by sending in the test packets into Wireshark.
Displaying bounded number of packets in Wireshark	Using FFShark and Wireshark, managed to display multiple packets from FFShark in Wireshark if there was a limit given to the number of packets.
C version of packet reading	Wrote a C version of our packet reading script to improve performance.
Performance measurement of full system	Ran multiple experiments to get performance data of our full system with different scenarios.
Verifying functionality of full system	Developed infrastructure to test our complete system and ensure all data remained uncorrupted.

Contributions of Tobias Rozario

The following table summarizes all of Tobias' contributions throughout the project.

Contribution	**Description**
Libpcap and Project Requirements Research	Researched *libpcap* interface, a method of interfacing Wireshark with custom hardware. Researched requirements to find performance and usability goals.
Initial Arm to Wireshark Packet Transfer & Formatting	Setup initial code to format a random packet and display it on Wireshark via sshdump from a remote machine.
Python code for sending packets into FFShark	Created Python code for randomly sending a set of test packets to FFShark.
Enabling user filtering instructions in Python	Enabled project to take in filtering instructions from users and then sending those filters to FFShark.
Enabled multiple packet transfer for end to end testing	Enabled multi-packet transfer between Wireshark and FFShark by letting the code send and receive packets infinitely. Also ensured that multiple resources do not access FPGA simultaneously to prevent crashes.
Python Code Speed Initial Measurements with profiling code	Measured our Python code speed with profiling code. Performed experiments for different packet sizes.
C library for ARM to FPGA communication	Created C library for ARM and FPGA communication, built off an existing Python library.
C Version of FFShark Packet Sending and C Code Initial Measurements with profiling code	Created C code for sending a set of test packets to FFShark in hopes of improving our packet filtering performance. Also set up instrumentation code in C to measure speed and discovered that it gave us a 20x speed upgrade with respect to the Python version.
Enabling user filtering instructions in C	Enabled taking in filtering instructions from users in the C version of our project.

Acknowledgments (Author: Alexander Buck)

First and foremost we want to thank our supervisor Professor Paul Chow for his guidance and support during our project. We also wish to thank Marco Merlini who gave up his time to help define the project, provide us with many helpful tutorials and ideas for getting started, and for teaching us how to use FFShark that he along with Juan Camilo Vega developed. We are also indebted to Clark Shen for his quick fixes of any issues we faced with using the server equipment. Additionally, we are thankful for the insightful feedback we received on our documents and presentation by our administrator Inci McGreal as well as by our communication instructors Peter Latka and Mina Arakawa. Last but not least, we would like to thank our families and friends for their ongoing support.

Executive Summary (Author: Tobias Rozario)

FFShark, an open-source high speed packet filtering hardware tool built at the University of Toronto, needed a graphical user interface (GUI) to improve its usability. Our project connects the industry standard Wireshark GUI to FFShark.

Our design has 3 main subsystems: a verification block, the FFShark Driver, and the *sshdump* to Wireshark connection. The user runs Wireshark on a remote machine which uses sshdump to send commands to the FFShark Driver running on the ARM machine. The FFShark Driver reads filtered packets from FFShark, formats, and sends the packets to Wireshark via sshdump. The verification block sends test packets to FFShark.

Overall this project has been declared successful. After performing component level and full system level testing, it has been demonstrated that our design meets all functional requirements and most objectives. Key functional requirements were allowing users to send in filtering instructions into FFShark through Wireshark and having Wireshark display filtered packets from FFShark. These were all met. Usability objectives, which involve supporting multiple operating systems and limiting the number of steps required to run our design with Wireshark, were also met. Only the performance objective of displaying filtered packets on Wireshark at a rate of 1 Gbps was not met as our design ran at 59 Mbps.

For future teams continuing this project, the main goal is to improve performance as basic functionality criteria are met. One possible approach is to use a more optimized hardware interface connecting the FFShark Driver to FFShark. Our project currently uses the standard AXI Stream FIFO interface. Future teams could look into replacing this with the AXI DMA interface which provides higher data transfer rates.

Table of Contents:

1.0 Introduction (Author: Alexander Buck)

This report provides background on Wireshark, FFShark, and the motivation to connect these two tools. Further, it will provide a summary of our design, testing procedures, and results in meeting our requirements. Lastly, we conclude with potential future work on the project. This work was performed as part of our final year design course, ECE496.

1.1 Background and Motivation (Author: Alexander Buck)

Modern high speed data centers run at speeds greater than 100 Gbps. To aid in debugging any internal issues, a general purpose processor would be unable to perform packet filtering at these speeds. Thus, custom hardware is required for this filtering task. Although commercial solutions exist [1], [2], FFShark was developed by J.C. Vega and M.A. Merlini at the University of Toronto to provide an open source FPGA based approach [3]. A block diagram is shown in Figure 1. It acts as a passthrough block for packets while also outputting filtered packets on a third port for further analysis.

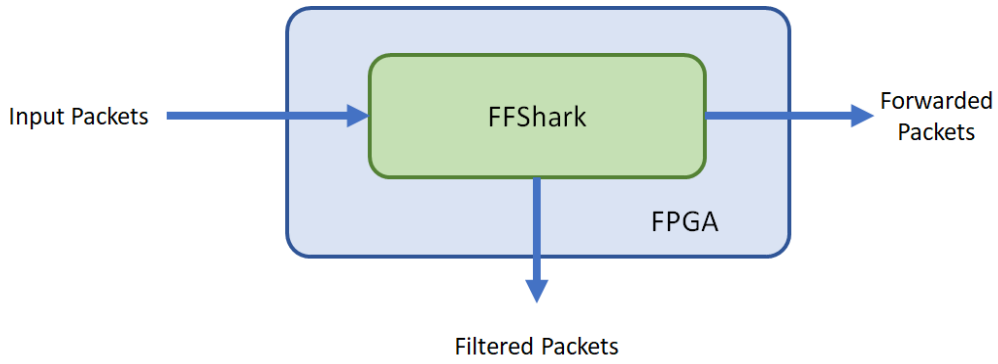


Figure 1: A block diagram of FFShark. Packets are input in one end and forwarded through the other end. Packets matching a filter are output on a third port.

FFShark did not originally have GUI support to provide an easy interface for users. To address this gap, we aimed to connect it to Wireshark. Wireshark is the de facto industry standard network analysis software [4]. It provides a GUI and additional features for debugging networks.

1.2 Project Goal and Requirements (Author: Tobias Rozario)

Our goal is to connect FFShark to Wireshark. The two functions our design must achieve are in Table 1. To measure the performance of our project, we developed 6 objectives shown in Table 2. Our constraints are in Appendix C. They were not included in the main report as they relate to the setup we were provided by our supervisor. They only limited our design space but did not directly influence our design decisions.

Table 1: Functions defining the basic criteria of the design

ID	**Function**	**Explanation**
F1	Display captured filtered packets from FFShark on Wireshark	Wireshark needs to be used to display filtered packets to the user.
F2	Send filters to FFShark via Wireshark	The design needs to allow Wireshark to send filters to FFShark.

Table 2: Objectives defining the goals of the design

ID	Objective	Explanation
O1	Limit the number of clicks from the Wireshark main menu in order to view FFShark captured filters (at most 2 clicks)	Two clicks were chosen because users perform 1 click on the GUI to select the capture interface (Wi-Fi, Ethernet, LAN etc.) [4]. 1 extra click should be required at most to see FFShark captured data.
O2	Limit the number of extra scripts to run per session when running our design with Wireshark (at most 3)	A maximum of 3 scripts was chosen since many open-source designs interfacing Wireshark to custom hardware have 2-4 [5], [6].
O3	Limit the number of scripts to run or plugins to install during the initial setup (2 at most)	Initial setup is the installation steps performed when running our design with Wireshark for the first time. Many custom open-source Wireshark based projects require only 1-2 plugins/scripts to be installed/run [5], [6].
O4	Remove the need for users to access the Linux machine running on the ARM machine	Currently, users need to remotely access the Linux machine running on the ARM machine to program the FFShark bitstream onto the FPGA. The goal is to provide an abstraction such that users do not need to be aware of these steps.
O5	Support 1 Gbps rate of captured filters	FFShark has a 1 Gbps Ethernet port connected to the ARM machine [3]**.** The goal is to use the entire bandwidth of the port to send filtered data.
O6	Support several operating systems (minimum of 2).	Common operating systems are Windows, MacOS and Linus. The goal is to support at least 2 of these operating systems to support more users.

2.0 Final Design Overview (Author: Tobias Rozario)

Our implementation allows Wireshark running on a remote machine to communicate with FFShark on an FPGA through the ARM machine connected to FFShark. This is seen in our system block diagram in Figure 2. *Sshdump* establishes a connection between the remote machine running Wireshark and the ARM machine. A user then sends packet filtering instructions to the FFShark Driver via sshdump. The driver compiles and sends the user's filtering instructions to FFShark. Next, the verification blocks simulate network packet traffic by generating and sending random packet data to FFShark. FFShark then sends out filtered packets to the FFShark Driver. The FFShark Driver formats packets into the PCAP file format which is compatible with Wireshark and sends the file to Wireshark through sshdump. Finally, the filtered packets are displayed on the Wireshark GUI.

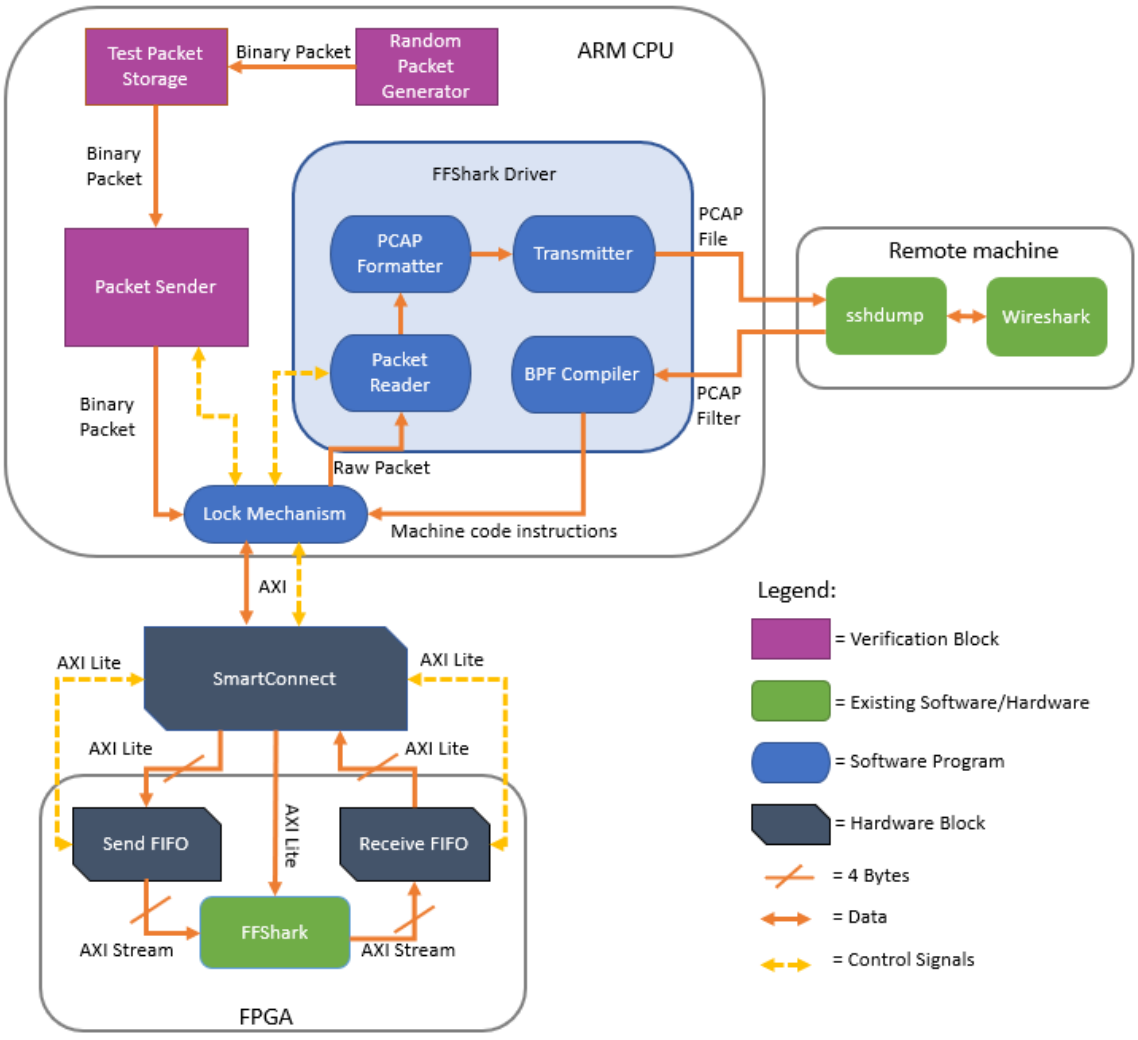


Figure 2 - A system block diagram of the design.

2.1 Hardware (Author: Kanver Bhandal)

The hardware portion of our project consists of four components: the SmartConnect, the Send FIFO, the Receive FIFO, and FFShark. All of our hardware components use a variant of the AXI protocol to communicate with each other. AXI is a high performance hardware communication protocol. Refer to Figure 3 to see a visual representation of how they connect to each other and the AXI interfaces they use to communicate with one another.

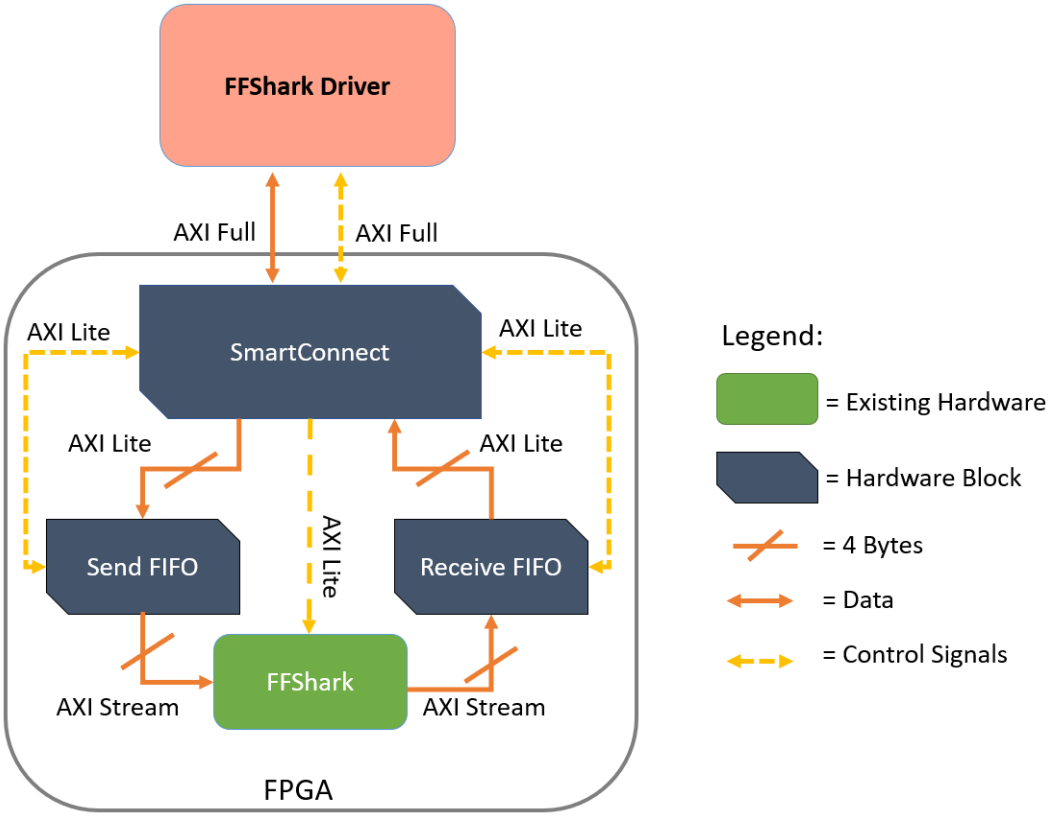


Figure 3 - Hardware Interface between FFhark and FFShark Driver

2.1.1 SmartConnect (Author: Kanver Bhandal)

The SmartConnect is needed in our design as it serves as a way for programs running on the ARM chip to communicate with hardware implemented on the FPGA. As we use AXI Stream FIFOs, we need to be able to send and read data from the FIFOs and the only way to do so is through the SmartConnect from the ARM chip via memory mapped I/O. FFShark also requires the SmartConnect in our design so that our FFShark Driver can program FFShark with filters and enable/disable it depending on what filter the user enters into Wireshark. The SmartConnect converts memory read and writes to certain memory addresses (the memory mapped I/O

addresses) on the ARM chip into AXI Full or AXI Lite signals on the FPGA, as shown in Figure 3.

2.1.2 Send and Receive FIFO (Author: Kanver Bhandal)

The Send FIFO and Receive FIFO are both AXI Stream FIFOs. They are used to send in data to FFShark and read out filtered packets from FFShark. A FIFO is used as the ARM chip has other processes running on it so it cannot poll on the FIFO and immediately read out a packet when a packet has been filtered. Hence, we use a FIFO so that multiple packets can be filtered out of FFShark and be stored until the ARM chip is ready to read them. The FIFOs also have an AXI Lite interface which allows them to be accessed by the ARM chip through the SmartConnect. This means programs running on the ARM chip can now access the filtered packets from FFShark and then send them to Wireshark. The FIFOs send and read data using an AXI Stream interface as that is the only AXI interface used by FFShark for filtering packets. However, communication to the ARM chip uses AXI Lite as the ARM chip can only use AXI Full or AXI Lite interfaces [7], [8]. Refer to Figure 3, for a visual representation of the AXI connections.

The send FIFO is only used to help with testing so that we can send in data to FFShark. In a real use case, FFShark would be filtering from an Ethernet connection and only a receive FIFO would be required as the ARM chip does not need to send in packets.

2.1.3 FFShark (Author: Kanver Bhandal)

FFShark is a high speed hardware implementation of a Berkeley Packet Filter (BPF) used for filtering network packets. It receives packets via an AXI Stream interface and outputs them using an AXI Stream interface. The reason it uses AXI Stream is that AXI Stream is a point to point protocol that has very low overhead allowing for a very high rate of data transfer [9]. AXI Stream is needed to filter networks running at 100 Gbps [3]. It also has an AXI Lite interface that is used to program FFShark with BPF filters and enable/disable it. Refer to Figure 3, for a visual representation of the AXI connections to FFShark.

2.2 FFShark Driver (Author: Tobias Rozario)

The FFShark Driver is responsible for reading out filtered packets from FFShark and sending them to Wireshark so that the packets will be displayed. It consists of four subcomponents: PCAP Formatter, Transmitter, Packet Reader and BPF compiler which are seen in Figure 4. In addition, transactions between the Driver and FFShark need to go through the Lock Mechanism. The Lock Mechanism prevents multiple resources from simultaneously accessing the FPGA to avoid crashes. Further details on subcomponents can be found in the subsections of this section.

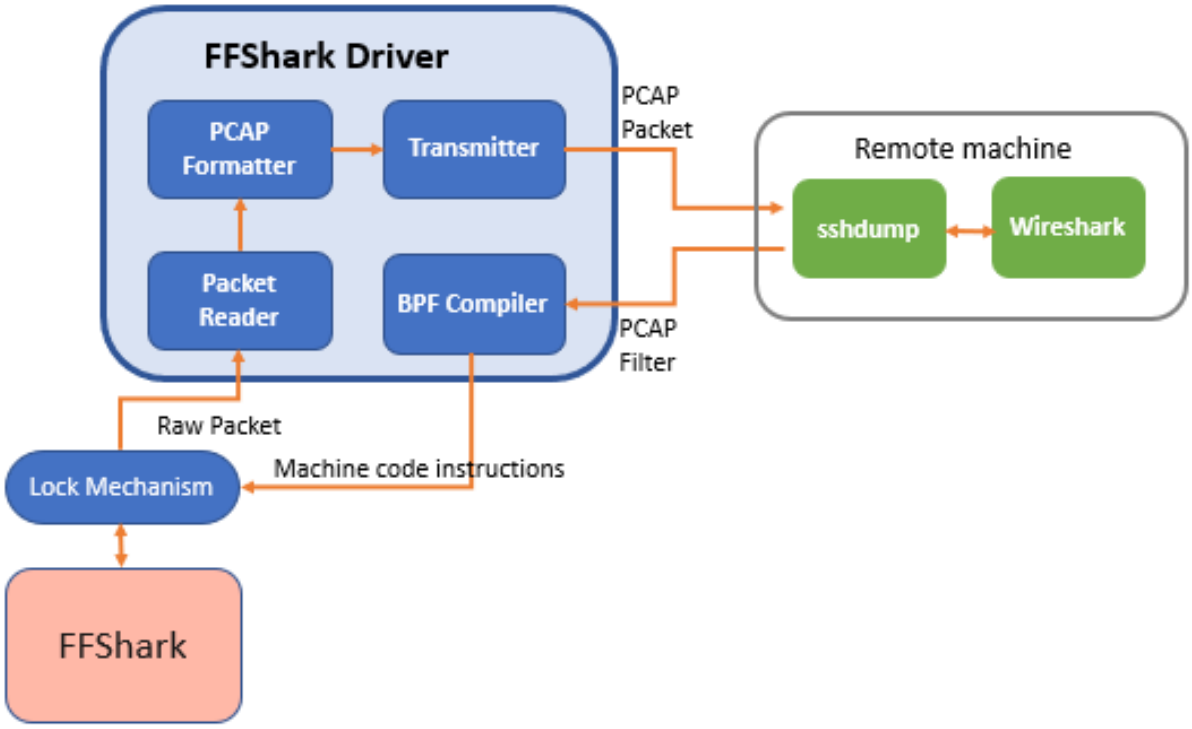


Figure 4 - FFShark Driver Submodule Diagram

2.2.1 BPF Compiler (Author: Tobias Rozario)

The driver receives the user's filtering instruction from the PCAP filter via sshdump. The BPF compiler compiles this filter into machine code and sends it to FFShark. FFShark uses the machine code to configure its packet filter.

2.2.2 Packet Reader (Author: Tobias Rozario)

The Packet Reader reads raw filtered packets from FFShark and sends the data to the PCAP formatter.

2.2.3 PCAP Formatter (Author: Tobias Rozario)

The PCAP formatter formats packets into PCAP files which are compatible with Wireshark. A PCAP file consists of a global file header, headers for each of the individual packets and the data of all the packets [10]. This can be seen in Figure 5. The global file header contains information such as a magic number, and major and minor version numbers. The magic number helps distinguish the start of the file, while the major and minor numbers dictate the version number of the PCAP formatting system. The individual packet headers include information such as timestamp and length of the packet. Each packet header will have its corresponding packet data next to it.

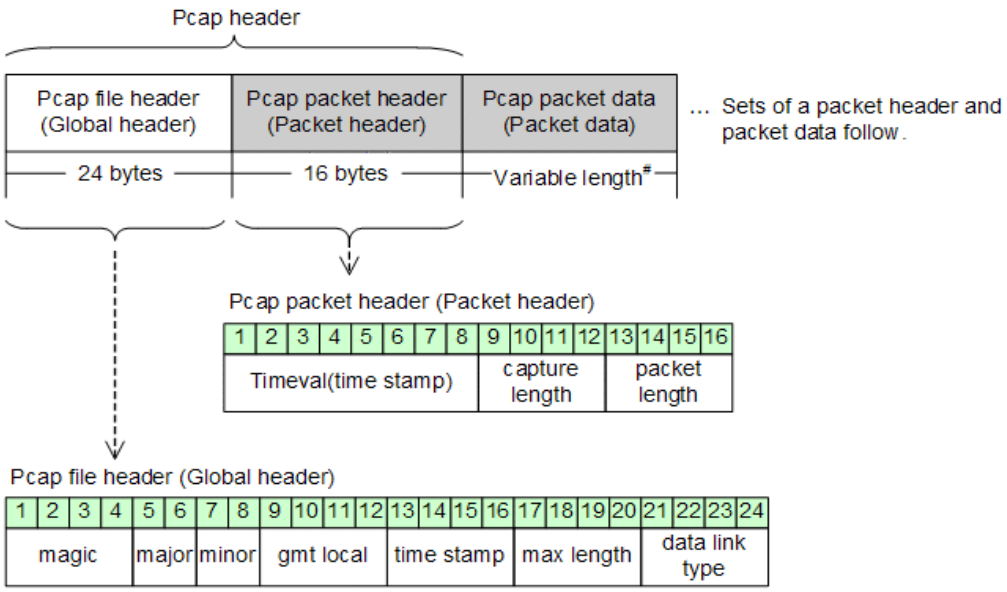


Figure 5 - PCAP Formatting [11]

2.2.4 Transmitter (Author: Tobias Rozario)

The transmitter prints the PCAP formatted packet onto the terminal so that the packet can get transferred to Wireshark via sshdump. Wireshark then displays the packet.

2.3 Verification Blocks (Author: Alexander Buck)

To test our design, we developed special verification blocks shown in Figure 6. These were used to send data into FFShark and simulate real network traffic. The Random Packet Generator is a

Python script that uses *scapy* to create packets with random payloads for different network protocols [12]. We manually generated the packets and stored them in the Test Packet Storage. From there, we used a custom developed script called Packet Sender to send packets into FFShark.

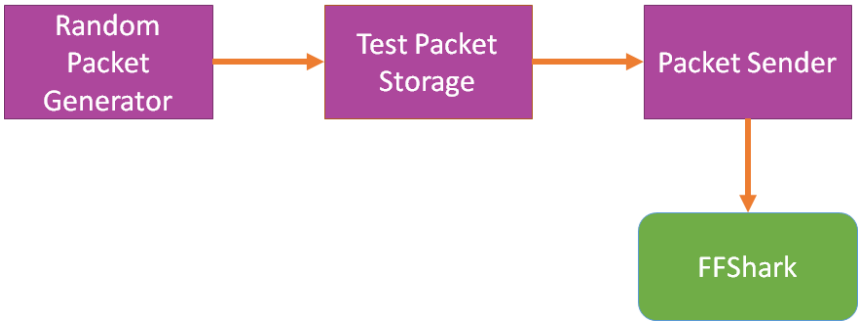


Figure 6: The verification subsystem used to simulate network traffic.

2.4 Wireshark (Author: Alexander Buck)

Wireshark is used as the interface for the user. It is used to start the capturing of network traffic and display captured packets on its GUI. To connect to the FFShark driver, we used an extension of Wireshark called sshdump.

2.4.1 sshdump (Author: Alexander Buck)

Sshdump can connect to a remote machine and use it as the network capture interface. We use sshdump to connect from the remote machine running Wireshark to the ARM running the FFShark driver. It provides the connection such that our driver only needs to output data to the terminal which is then transferred to Wireshark.

2.5 Assessment (Author: Kanver Bhandal)

In the beginning of our project, we divided our project into appropriate sized blocks. We identified which components needed to be created in hardware and which components needed to be developed in software. This naturally allowed us to partition our project such that one person would be responsible for the hardware interface and the other two would be responsible for the software components. For the software components, we further partitioned the work by having one person focus on the FFShark driver and the other focus on creating a remote connection between the ARM chip and Wireshark. This allowed us to work in parallel and be more productive.

Our approach to our design has been methodical. We first focused on getting basic functionality working and then on performance. This mindset led us to use AXI Stream FIFOs as our hardware interface since it is the simplest one possible. The FIFOs use the simplest AXI protocol (AXI Lite) and we had working demonstrations of how to use the FIFOs from Marco Merlini. For software, we first wrote our FFShark Driver in Python as it allowed us to rapidly write our code and had many libraries to reduce our workload. This approach worked very well for us as it provided us with a minimum viable product early on into our design and from then on we could focus on improving the performance of our solution.

After successfully creating a design that met our functional requirements and most objectives, we decided to improve the performance of our design by profiling our code for bottlenecks. If we had blindly started to optimize, or done premature optimization, we would not have reached our current performance numbers as we would have most likely been optimizing code that had no effect on our performance. Profiling our code was more work initially, but it allowed us to effectively target bottlenecks and greatly improve our performance. We found that the PCAP formatting was taking the majority of our time after writing the FFShark Driver in C.

One lesson learned is to record our results more thoroughly and frequently. Throughout our design process we did not note all results in formal manners. This culminated in us having to re-run several of our tests or measurements for writing our project documents which prevented us from working more on our actual design. In the future, we would aim to document and record

all our results more thoroughly and often so that we do not need to re-run experiments or tests as often.

Another issue that occurred during our design process was scope creep or not adhering to our scope we decided in our project proposal. We had decided to not fix any bugs in FFShark as our project was only concerned about connecting FFShark to Wireshark. However, as we kept working on our project we discovered bugs within FFShark. We spent a non-trivial amount of time investigating, root causing, and developing a fix for the FFShark bugs which took time away from working on connecting FFShark to Wireshark. Had we quickly documented the bug as a FFShark bug and moved on, we would have had more time to spend on optimizing our design. In the future, we plan to adhere strictly to our scope so that we can focus on our design and not get distracted by unrelated problems.

Overall, our project was successful in achieving its main requirements. We focused on splitting the work evenly and ensured we could all work in parallel. Additionally, focusing on core functionality first allowed us to create a working product to demonstrate. We then systematically analyzed our design to improve its performance to create a well functioning final product.

3.0 Test and Verification Overview (Author: Alexander Buck)

We split our verification into two categories, functional and performance measurements in sections 3.1 and 3.2 respectively. For both of these categories, our testing plan consisted of testing each subsystem separately and then testing the fully integrated solution. This approach both aided our development timeline and gave us more confidence that our verification procedures were correct. In Table 3 we show how we tested our project requirements and their results.

Table 3: Verification Table

ID	Project Requirement	Verification Method	Verification Result and Proof
F1	Display filtered packets from FFShark on Wireshark	**TEST:** Semi-automated method to compare packets sent in to FFShark to packets received by Wireshark	**PASS**, all packets were received by Wireshark. See 3.1 for details
F2	Send filters to FFShark via Wireshark	**TEST:** Use Wireshark to send filters and ensure packets are filtered correctly	**PASS**, only filtered packets are received by Wireshark. See 3.1 for details
C1	Must use Wireshark packet analyzer to display FFShark	**REVIEW of DESIGN:** Wireshark is used	**PASS**, design uses Wireshark
C2	Must run Wireshark on an external workstation	**REVIEW of DESIGN:** Wireshark does not run on the ARM machine	**PASS**, Wireshark runs on separate machine
C3	Must use the Ethernet port from the ARM machine to connect to the internet	**REVIEW of DESIGN:** Ethernet is the only connection from the ARM machine to the internet	**PASS**, Ethernet port is used
C4	Must not utilize the *MUL* (multiplication), *DIV* (division), and *MOD* (modulo) filtering instructions	**TEST:** Send filters that use these instructions and report an error to the user	**UNTESTED**, common filters that were tested did not use those instructions so we did not have any errors

O1	Limit the number of clicks in Wireshark from main menu to view packets to 2	**REVIEW of DESIGN:** Manually count	**PASS**, using default values requires 2 clicks. Changing filters requires 3 clicks
O2	Limit the number of scripts to run per session to 3	**REVIEW of DESIGN:** Manually count	**PASS**, one script is run on ARM and one plugin (sshdump) is run on Wireshark
O3	Limit number of initial setup scripts/plugins to install to 2	**REVIEW of DESIGN:** Manually count	**PASS**, one script needs to be installed on the ARM chip and sshdump needs to be installed in Wireshark
O4	Remove the need for users to access the Linux machine running on the ARM machine	**REVIEW of DESIGN:** Manually verify no scripts or programs must be user initiated on the ARM machine	**PASS**, can start session from Wireshark if FFShark is loaded on FPGA
O5	Support 1 Gbps rate of captured filters	**TEST:** Measure and calculate the rate of packets displayed on Wireshark using Wireshark timestamps and packet byte sizes	**FAIL**, throughput is 59 Mbps. See Section 3.2 for detailed results
O6	Support minimum of 2 operating systems	**REVIEW of DESIGN:** Wireshark is able to connect to FFShark on 2 of the following operating systems: Linux, Windows, or MacOS	**PASS**, can use Windows or Linux. Untested for MacOS

3.1 Functional Verification (Author: Alexander Buck)

We performed and passed a full integration test of our project. This was done by writing out all packets sent into FFShark to a file. We then used Wireshark's export feature to export all the captured packets. We then performed a *diff* operation between these files to ensure all data was uncorrupted and received by Wireshark. Figure 7 shows this verification process.

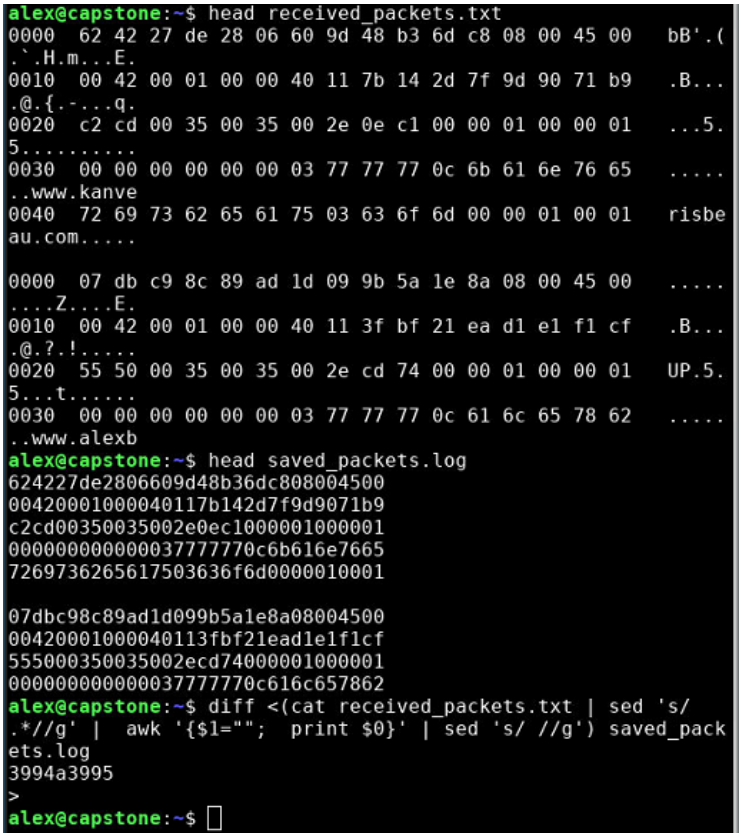


Figure 7: The first 10 lines of both the received_packets.txt (generated from Wireshark) and the saved_packets.log (generated from our send script) are shown. The diff command first converts the two files into the same format and then checks if there are any differences. No differences (apart from an additional new line character in the file due to formatting) are reported.

We additionally tested that filters were properly being programmed. We first set filters in Wireshark and then manually verified that only packets passing the filter were received. Figure 8 shows a screenshot of our system only accepting UDP packets.

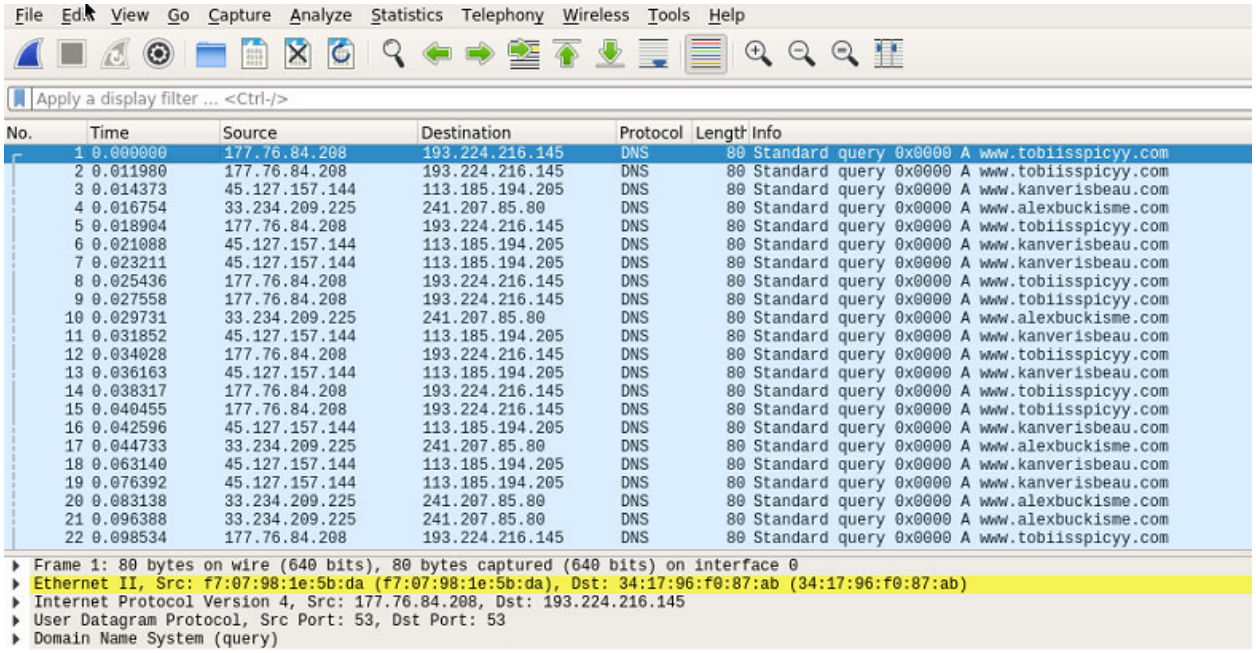


Figure 8: A screenshot of Wireshark with “udp” set as the filter for FFShark. Only UDP packets are accepted despite sending in UDP and TCP packets.

We tested functionality of individual blocks at finer granularity than just system level testing. The details for the hardware testing and FFShark driver are below.

3.1.1 Hardware (Author: Kanver Bhandal)

Table 4 summarizes tests performed to verify that all hardware components of the design were functional. As the SmartConnect connects to the FIFO and the FIFO connects to FFShark, testing the SmartConnect results in also testing the FIFO and FFShark as one whole sub-system.

Table 4: Hardware Components Testing

Component(s) Tested	**Verification Method**	**Result**
FFShark	**TEST:** Send packets into FFShark and check if FFShark outputs the packets using ILAs (Integrated Logic Analyzer, a tool to capture waveforms of the internals of the hardware)	**PASS**, refer to Figure 9

AXI Stream FIFOs	**TEST:** Send packets into the FIFO and and verify the data can be sent out of the FIFO using ILAs	**PASS,** refer to Figure 10
SmartConnect	**TEST:** Use the ARM chip to write to the memory mapped addresses for the Send FIFOs and confirm data is sent and received by reading the output registers of the Receive FIFO	**PASS,** refer to Figure 11

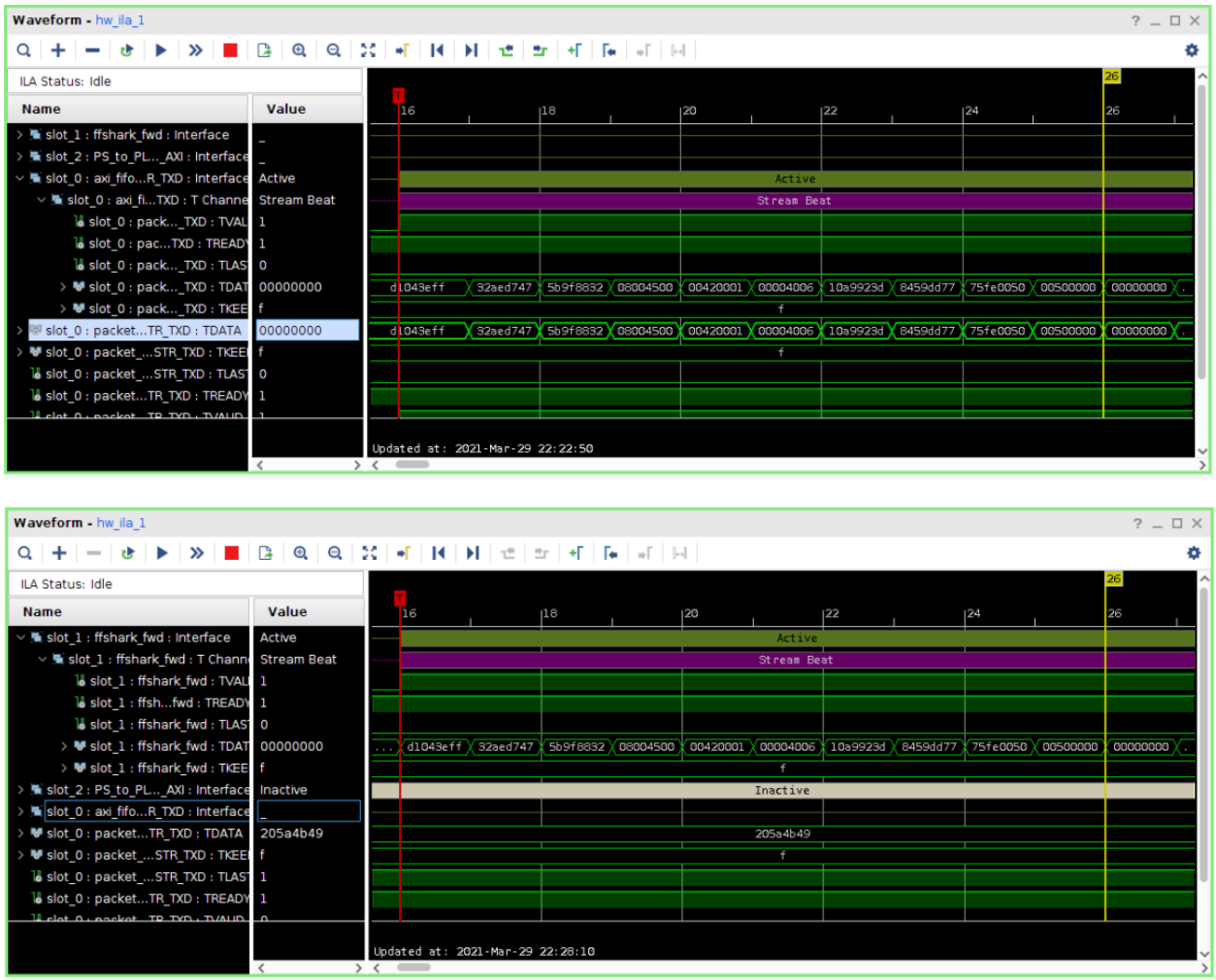


Figure 9 - The top waveform shows the Send FIFO sending data to FFShark and the bottom waveform shows the Receive FIFO accepting the output of FFShark The above screenshots show that the AXI Stream FIFOs can communicate with FFShark.

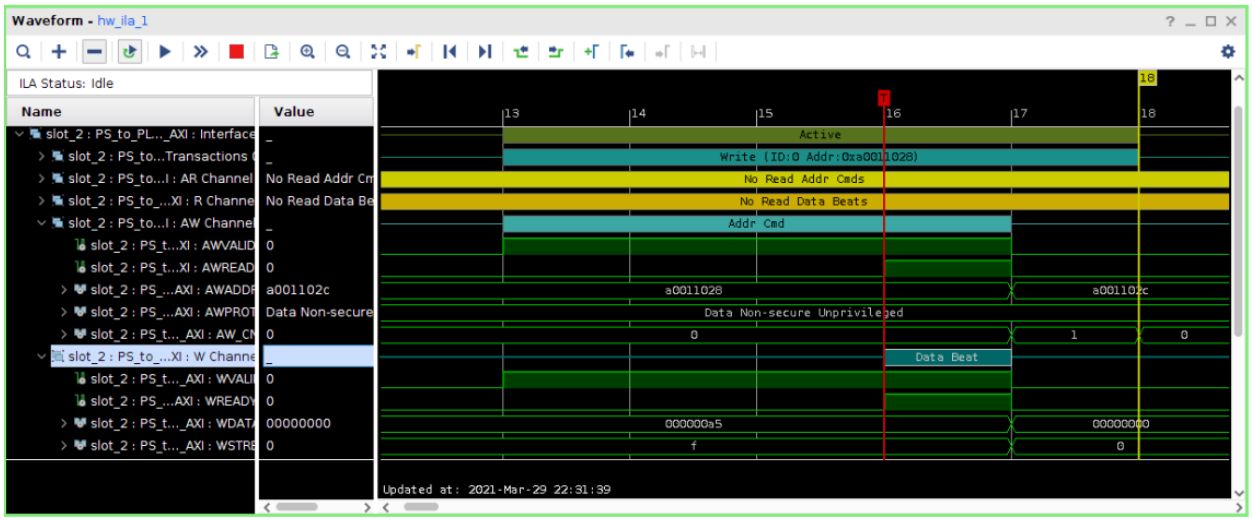


Figure 10 - The SmartConnect is able to write to the AXI Stream FIFO's registers to send and read packets, verifying that the connection between the two components is correct.

```
savi@mpsoc15:~/kanver/ffshark_tut$ python3 ffs
hark_send_packets.py --packets-directory ./FFS
hark_Wireshark_Integration/sample_packets/mult
iple_8 --num-packets 1 --debug
0xd1043eff
0x32aed747
0x5b9f8832
0x8004500
0x420001
0x4006
0x10a9923d
0x8459dd77
0x75fe0050
0x500000
0x0
0x5002
0x20009efc
0x4854
0x54502f31
0x2e312032
0x3030204f
0x4b0d0a0d
0xa545055
0x205a4b49
```

```
savi@mpsoc15:~/kanver/ffshark_tut$ python3 ffs
hark_read_packets.py --num-iterations 1 --debu
g
0xd1043eff
0x32aed747
0x5b9f8832
0x8004500
0x420001
0x4006
0x10a9923d
0x8459dd77
0x75fe0050
0x500000
0x0
0x5002
0x20009efc
0x4854
0x54502f31
0x2e312032
0x3030204f
0x4b0d0a0d
0xa545055
0x205a4b49
```

Figure 11 - The ARM Chip is able to use the SmartConnect to write packets into the Send FIFO, as shown in the left hand side screenshot. Once the Send FIFO receives the packet it sends it into FFShark which outputs the packet to the Receive FIFO. Then the ARM chip can read out the packets from the Receive FIFO, as shown in the right hand screenshot. Both the sent in packets and read out packets match.

3.1.2 FFShark Driver (Author: Tobias Rozario)

Table 5 summarizes the test results of the different components of the FFShark Driver.

Table 5 - FFShark Driver Components Testing

Component(s) Tested	**Verification Method**	**Result**
Formatter & Transmitter	**TEST:** Run the read script through Wireshark's sshdump and check if packets get displayed	**PASS**, refer to Figure 12
Packet Reader	**TEST:** Send packets into FFShark, then run the read script on terminal and check if packets get displayed on terminal	**PASS**, refer to Figure 13 and 14
BPF Compiler	**TEST:** Send in filtering instructions to FFShark Driver and check the correct filtered packets get sent out by FFShark	**PASS**, refer to Figure 15

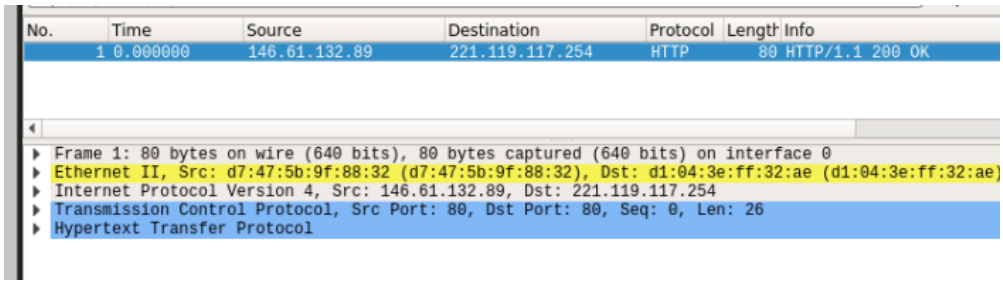


Figure 12 - Packet displayed on Wireshark to test the Formatter and Transmitter

```
savi@mpsoc15:~/kanver/ffshark_tut$ python3 ffshark_send_packets.py --packets-directory packets/ --num-packets 1
0
0
4
num vacant words ::: 16380
0xabcd3567
0
0xdfea8769
0
0x45671234
0
0xaabbdeed
0
16
0
```

Figure 13 - Terminal output showing 16 byte test packet sent to FFShark

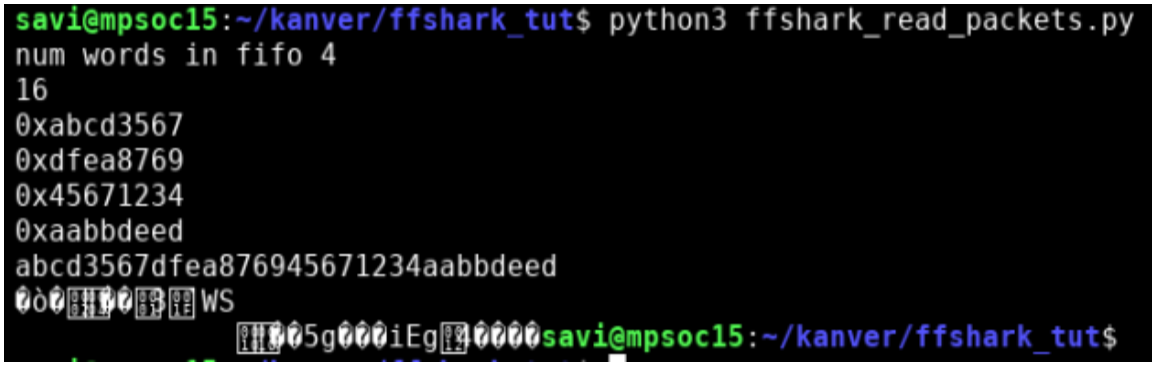


Figure 14 - Terminal output shows that a 16 byte packet read from FFShark matches the corresponding 16 bytes sent in Figure 13 exactly.

First set filter to UDP packets only, then send a TCP packet.

```
savi@mpsoc15:~/tobias/ffshark_tut$ python3 comp_and_send_main.py
savi@mpsoc15:~/tobias/ffshark_tut$ python3 ffshark_send_packets.py --num-packets=1 --packets-directory=s
mple_packets/tcp_packets/
```

Readout FFShark packet from the FIFO. There are no packets in the FIFO because the filter is only for UDP packets. Thus, the TCP packet was discarded by FFShark's filter.

```
savi@mpsoc15:~/tobias/ffshark_tut$ python3 ffshark_read_packets.py --num-iterations=1
num words in fifo 0
```

Send a UDP packet

```
savi@mpsoc15:~/tobias/ffshark_tut$ python3 ffshark_send_packets.py --num-packets=1 --packets-directory=sa
mple_packets/udp_dns/
0
```

UDP packet gets read from FFShark as the filter does not discard it.

```
savi@mpsoc15:~/tobias/ffshark_tut$ python3 ffshark_read_packets.py --num-iterations=1
num words in fifo 20
80
0x9c1610bc
0x2b460dac
0x68867be1
```

Figure 15 - Execution stream showing that filtering functions properly

3.2 Performance Verification (co-authors: Alexander Buck and Tobias Rozario)

Our key performance measurements involved the overall throughput from FFShark to Wireshark. All detailed proof of measurements may be found in Appendix D. Multiple scenarios were tested using Python and C both with and without locking. We used the “Capture File Properties” feature in Wireshark to automatically determine the throughput of the end-to-end system. Results are shown in Table 6. Locking is only required in our setup because we need to manually send in test packets using the same bus that is used for reading packets. By preloading FFShark with packets, we can simulate a real life use case without locking. In this case we achieve a throughput of 59 Mbps using C.

Table 6: Throughput of our code in C and Python with and without locking.

Language	**Locking?**	**Throughput (Mbps)**
C	Yes	23
Python	Yes	0.087
C	No	59
Python	No	0.091

We performed further measurements of different subsystems of our design. This was used to determine where we may face bottlenecks and where our performance is limited by the hardware we are provided. The results are shown in Table 7. We simulated “perfect” hardware by using pre generated random data directly on the ARM chip that is then read out by Wireshark. The maximum transfer rate from FFShark to ARM was determined by reading the packet data on FFShark but performing no other operations on it. Thus our software would not be bottlenecking the performance. The SSH throughput was measured by transferring 100 MB of random data using *netcap* between the remote machine and the ARM. We then used Wireshark to snoop on the data to find the transfer rate.

Table 7: Additional measured throughputs of our subsystems

Scenario	**Locking?**	**Throughput (Mbps)**
C code with “perfect” hardware	Yes	105
C code with “perfect” hardware	No	230
Maximum transfer rate from FFShark to ARM	No	95
Maximum SSH throughput	No	464

4.0 Summary and Conclusion (Author: Kanver Bhandal)

Our goal was to connect FFShark to Wireshark. To be successful, our design had to perform two functions. First it had to display filtered packets from FFShark on Wireshark. Second, it had to allow the user to be able to set filters in Wireshark that would be programmed into FFShark. These two functions were met by our design as a user on Wireshark can use sshdump to launch our FFShark driver remotely on the ARM Chip to read out filtered packets from FFShark. Additionally, the user can provide a filter to our FFShark Driver which will program FFShark with the filter. All our objectives related to enhancing the usability of our design were met but we did not meet the objectives for supporting filtering at 1 Gbps and not allowing multiply, division, and modulo instructions inside a filter.

We ensured that our design functioned correctly by sending a large number of packets into FFShark and verifying that Wireshark displayed all of them. This test was passed. Additionally, before performing end to end testing, we verified each component individually to ensure it worked correctly. The component level testing and system level testing ensured that our design was functional.

For testing the performance of our design, we measured the throughput of our system using Wireshark's built in analytics to measure packet throughput. We considered the non-locking version of our code as our performance benchmark because that is the version of our code that would be used in a real scenario with FFShark filtering packets from a 100 Gbps network. We discovered that the initial version of our design running Python transferred packets at a rate of 0.091 Mbps. As Python is not a high performance programming language, we rewrote our FFShark Driver in C and performed other optimizations to the PCAP formatting and reached a throughput of 59 Mbps. Our optimizations resulted in a 648x improvement in performance.

Our design is fully functional and has adequate performance, but there is future work that could improve performance. We conducted a test to measure the throughput of only our FFShark Driver with "perfect" hardware and discovered our FFShark Driver by itself could have a throughput of 230 Mbps. We found that we were being limited by our hardware interface as the AXI Stream FIFOs were limited to a throughput of 95 Mbps. This shows our design is

bottlenecked by our hardware interface. This is significant as the throughput of our design limits how general of a filter can be programmed into FFShark. If our design has higher throughput it allows FFShark to be able to output more packets that match the filter. A more general filter has more packets being accepted by FFShark which requires our design to have a higher throughput so it can read the filtered packets out of FFShark. One way to improve our hardware interface is to use an AXI DMA instead of AXI Stream FIFOs. A DMA is faster than an AXI Stream FIFO as it directly writes into the ARM chip's memory.

Our work is useful for industry and academia. Current existing 100 Gbps packet filters are expensive and proprietary. FFShark is an open source 100 Gbps packet filter. Researchers and industry professionals who require high speed packet filtering are now able to use FFShark to analyze and debug high speed networks. Our addition of adding a GUI, Wireshark, to FFShark allows FFShark to be used by researchers and industry professionals who are not familiar with FPGAs, or low level coding/scripting. Our project made FFShark easier to use, set up, and increased its chances of being used in industry and academia as a packet filtering solution.

5.0 References (Co authors: Tobias Rozario, Kanver Bhandal)

[1] “10/25/40/100G Packet Broker with PCAP Filtering,” product Brief. [Online]. Accessed September 21, 2020. Available: https://www.bittware.com/102540100g-packet-broker-with-pcap-filtering/.

[2] “Full Line Rate Sustained 100Gbit Packet Capture,” product Brief. [Online]. Accessed September 21, 2020. Available: https://www.fmad.io/products-100G-packet-capture.html.

[3] C. Vega, M. Merlini and P. Chow, “FFShark: A 100G FPGA Implementation of BPF Filtering for Wireshark,” in 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). [Online]. IEEE, 2020, Accessed September 21, 2020. Available: https://www.fccm.org/past/2020/proceedings/2020/pdfs/FCCM2020-65FOvhMqzyMYm99lfeVKyl/580300a047/580300a047.pdf.

[4] Wireshark, “Wireshark User’s Guide Version 3.3.1,” 2020, [Online]. Accessed September 21, 2020. https://www.wireshark.org/docs/wsug_html_chunked/.

[5] A. Cardigliano, “PF-RING - Wireshark Extcap”, 2020, [Online]. Accessed September 21, 2020. https://github.com/ntop/PF_RING/tree/dev/userland/wireshark/extcap.

[6] N. Bowden, “WLAN-Pi - WLANPiShark2”, 2019, [Online]. Accessed September 21, 2020. https://github.com/WLAN-Pi/WLANPiShark2

[7] "SmartConnect v1.0 LogiCORE IP Product Guide SmartConnect v1.0 4 PG247 February 3, 2020 Product Specification - [PDF Document]", vdocuments.site, 2021. [Online]. Available: https://vdocuments.site/smartconnect-v10-logicore-ip-product-guide-smartconnect-v10-4-pg247-february-3.html. [Accessed: 30- Mar- 2021].

[8] O. Media, "Interfacing embedded FPGAs with ARM buses - Embedded Computing Design", Embedded Computing Design, 2021. [Online]. Available: https://www.embeddedcomputing.com/application/misc/arm-bus-interfaces-for-embedded-fpgas. [Accessed: 30- Mar- 2021].

[9] "AXI Reference Guide", *Verien.com*, 2021. [Online]. Available: http://www.verien.com/axi-reference-guide.html. [Accessed: 30- Mar- 2021].

[10] "Development/LibpcapFileFormat - The Wireshark Wiki", *Wiki.wireshark.org*, 2021. [Online]. Available: https://wiki.wireshark.org/Development/LibpcapFileFormat. [Accessed: 30- Mar- 2021].

[11] "Overview of HTTP packet input", *Itdoc.hitachi.co.jp*, 2021. [Online]. Available: http://itdoc.hitachi.co.jp/manuals/3020/30203V0200e/BV020176.HTM. [Accessed: 30- Mar- 2021].

[12] P. community., "Scapy", *Scapy*, 2021. [Online]. Available: https://scapy.net/. [Accessed: 30- Mar- 2021].

[13] GitHub, “Pricing”, [Online]. Accessed November 22, 2020. https://github.com/pricing

[14] Start, “Internet Packages”, [Online]. Accessed November 22, 2020. https://www.start.ca/services/high-speed-internet#packages

[15] Xilinx, “Vivado Design Suite - HLx Editions”, [Online]. Accessed November 22, 2020. https://www.xilinx.com/products/design-tools/vivado.html#buy

[16] Best Buy, “ASUS ZenBook 14" Laptop”, [Online]. Accessed November 22, 2020. https://www.bestbuy.ca/en-ca/product/asus-zenbook-14-laptop-dark-royal-blue-intel-core-i7-10510u-512gb-ssd-16gb-ram-windows-10-en/14470973

Appendices

Appendix A: Gantt Chart History (Author: Kanver Bhandal)

Below in Figure 16, is our most recent version of our Gantt Chart. We have completed all our tasks related to the functionality and testing of our project. The only incomplete tasks were some further potential optimizations to our project, such as using a DMA controller, or bug fixes to FFShark which were outside the scope of our project.

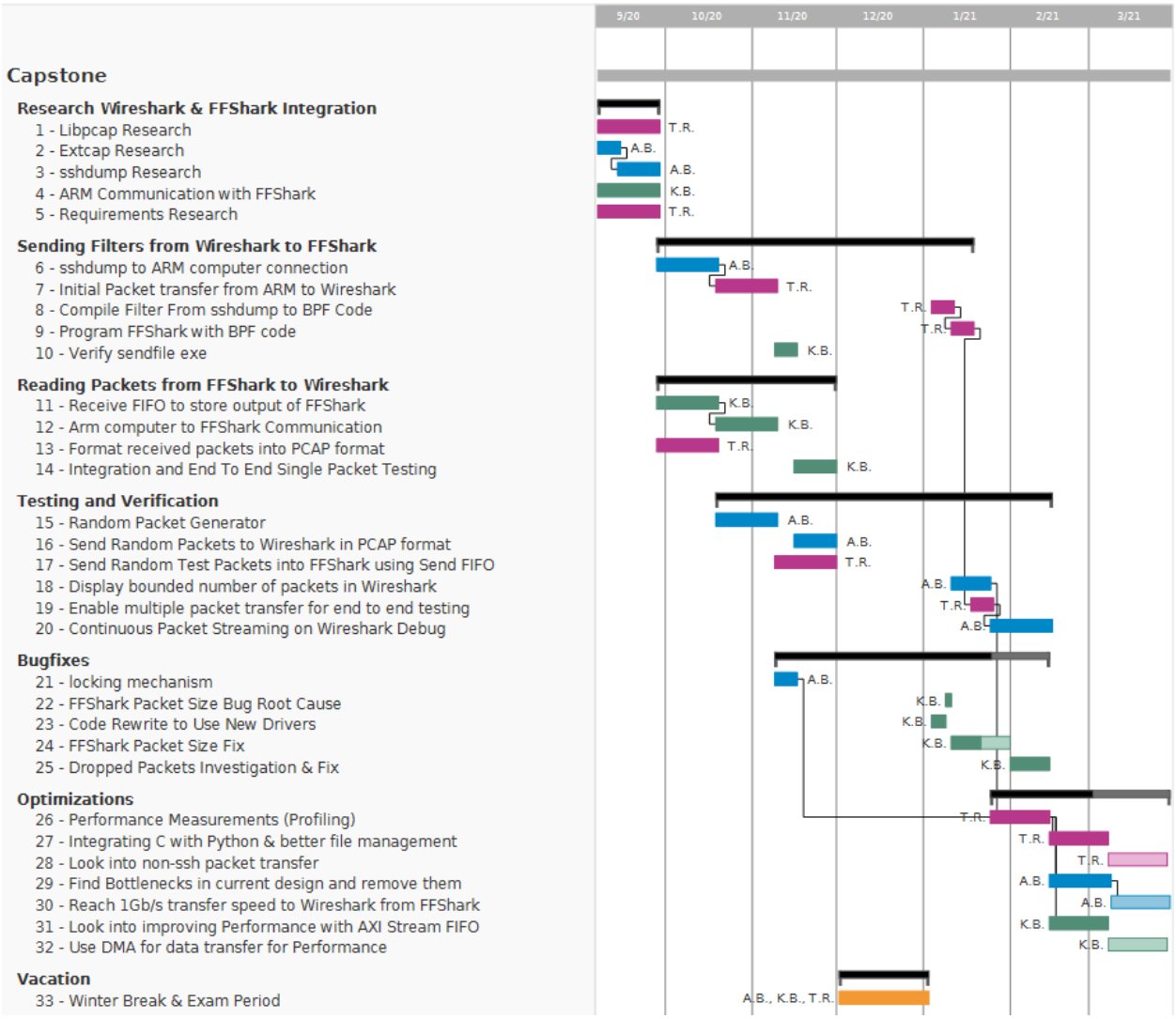


Figure 16 - Gantt Chart

Below in Figure 17, is our Gantt Chart from the Progress Report. The main tasks we had left at this point was final verification and testing, bug fixes, and optimizations.

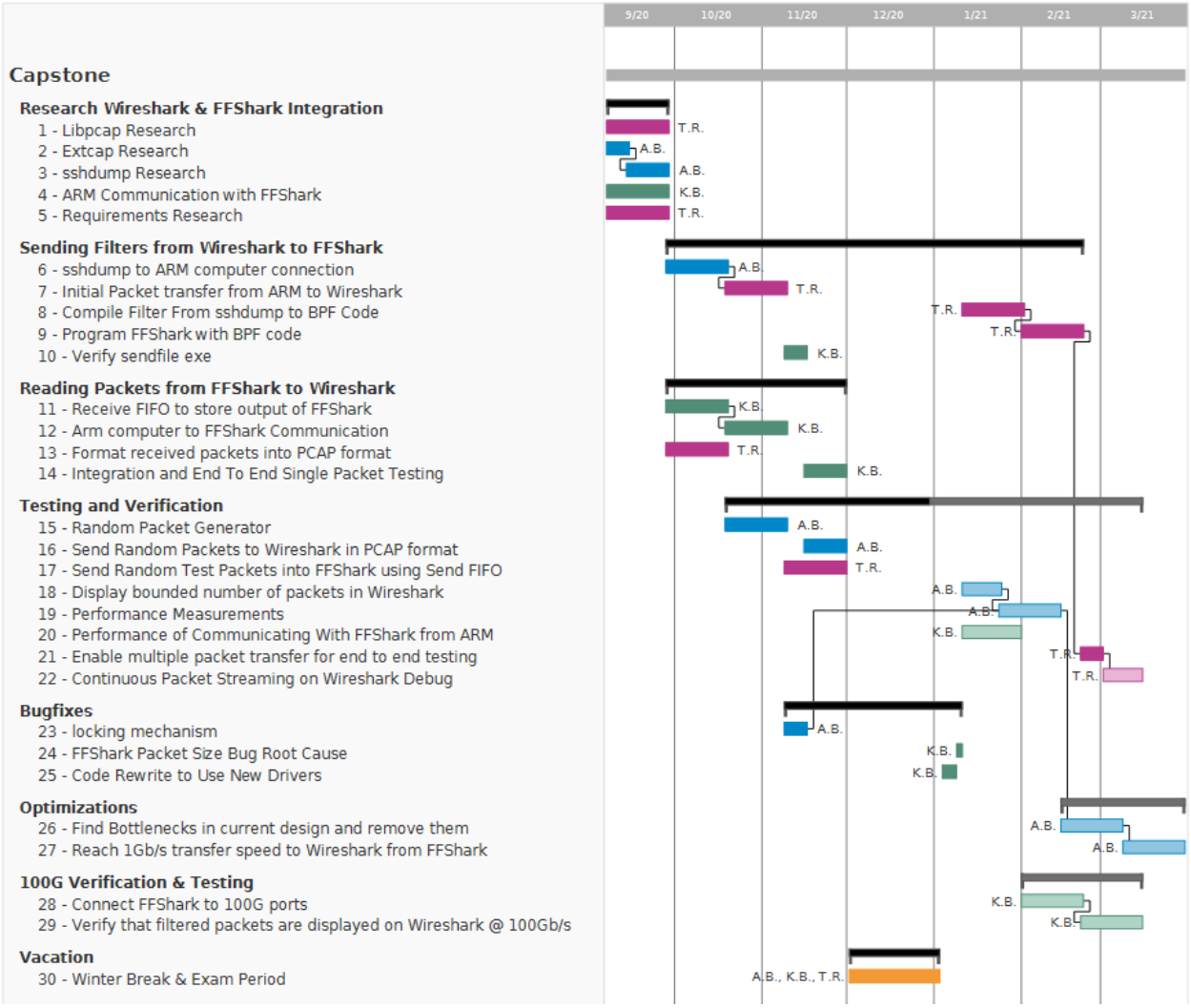


Figure 17 - Gantt Chart from Progress Report

Below in Figure 18 and Figure 19, is our Gantt Chart from our Project Proposal. At this point we finished our research and began working on implementing it.

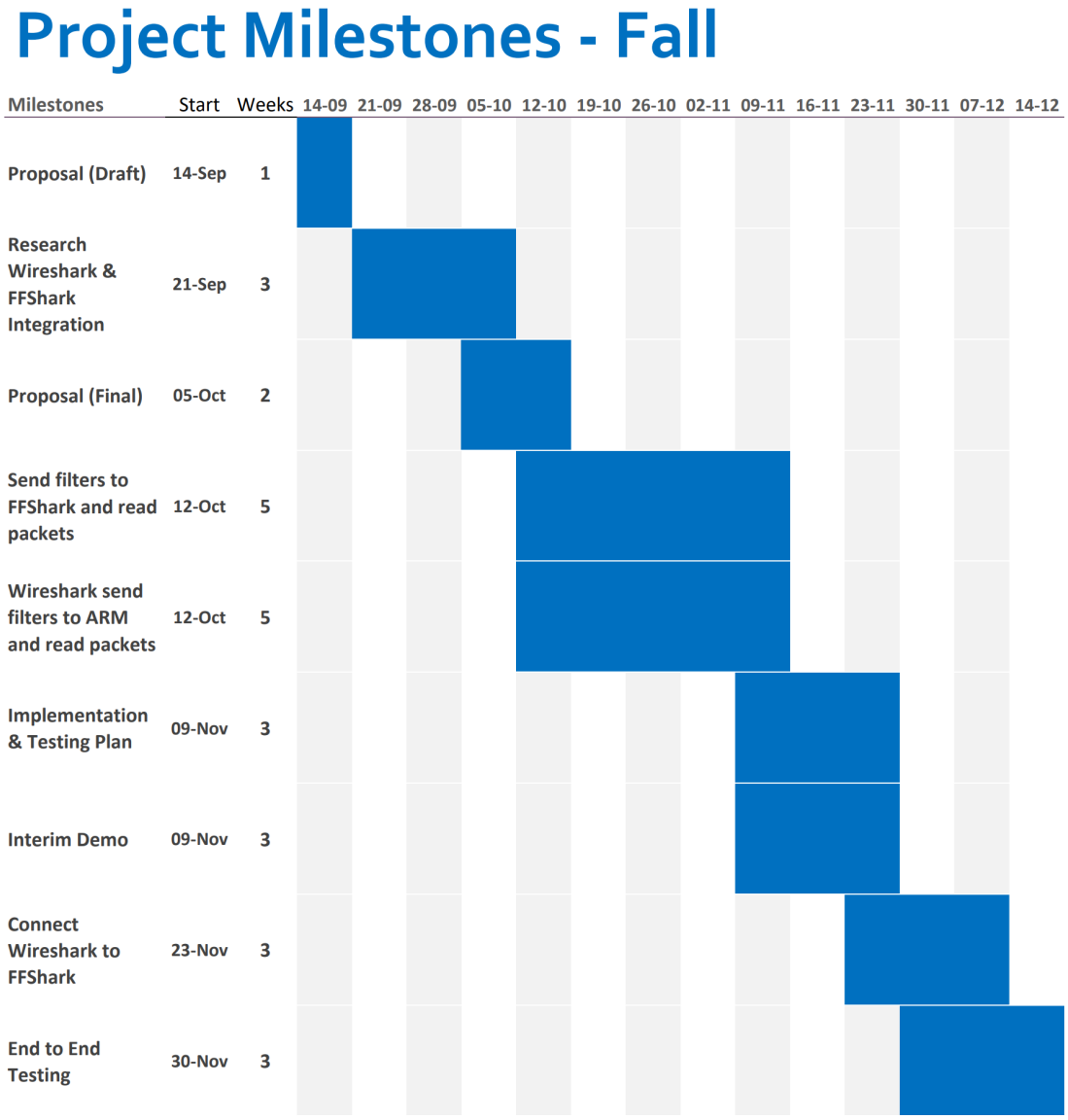


Figure 18 - Project Milestones for Fall Semester from Project Proposal

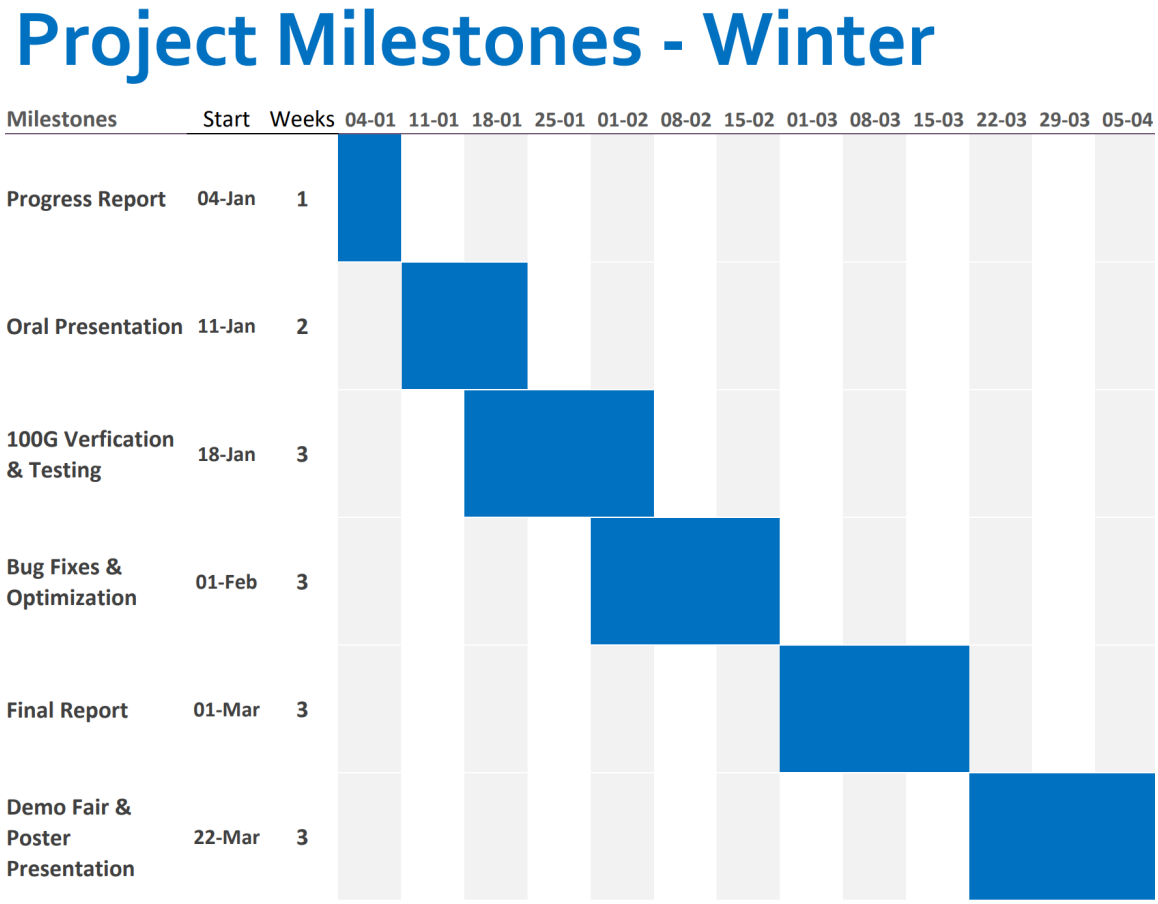


Figure 19 - Project Milestones for Winter Semester from Project Proposal

Appendix B: Financial Plan (Author: Alexander Buck)

The total cost of our project is $39,788 and is detailed in Table 8. Our financial plan remains unchanged from our implementation plan as we had no unexpected expenses that arose during the project. All values are listed in CAD. At the time of our capital expenses, an exchange rate of USD to CAD of 1.31 was used.

Table 8: The budget of our project.

Consumables and Services							
Item	**Monthly Cost/unit**	**Quantity (monthly if applicable)**	**Type (one time or monthly)**	**Total Cost (for 8 months)**	**Requires funding?**	**Kept/Paid for by student?**	**Source**
GitHub Pro	$ 4.00	3	Monthly	$ 96.00	n	y	[13]
Internet Plan	$ 55.00	3	Monthly	$ 1,320.00	y	y	[14]
Vivado Design Suite	$ 5,626.00	1	One time	$ 5,626.00	n	n	[15]
Total Consumables & Services				$ 7,042.00			
Total Requiring Funding				$ -			
Capital Equipment							
Item	**Cost/unit**	**Quantity**	**Total Cost**	**Requires funding?**	**Kept/Paid for by student?**	**Source**	
MPSoC	$ 8,646.00	1	$ 8,646.00	n	n	Pricing from supervisor	
Laptop	$ 1,500.00	3	$ 4,500.00	y	y	[16]	
Server	$ 1,600.00	1	$ 1,600.00	n	n	Estimate from supervisor	
Total Capital Equipment			$ 14,746.00	n			
Total Requiring Funding			$ -				
Labour							
Item	**Cost/unit**	**Quantity (hours)**	**Total Cost**	**Requires funding?**	**Source**		
Kanver Bhandal	$ 40.00	150	$ 6,000.00	n	30 weeks * 5 hours per week.		
Alexander Buck	$ 40.00	150	$ 6,000.00	n	$40/hour is typical entry level		
Tobias Rozario	$ 40.00	150	$ 6,000.00	n	wage in Toronto		
Total Labour (unfunded)		450	$ 18,000.00				
Total Cost of Project			$ 39,788.00				

Appendix C: Constraints (Author: Tobias Rozario)

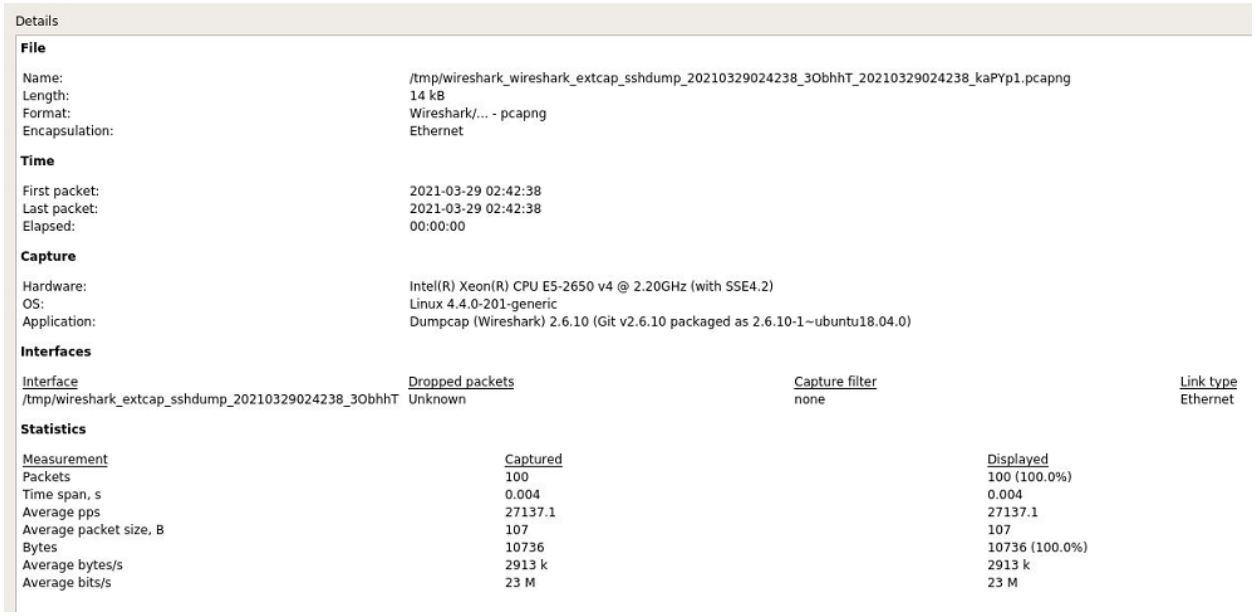
Table 9 below shows the constraints, given by our supervisor, that our design must follow.

Table 9: Constraints the design must follow

ID	Constraint	Explanation
C1	Must use Wireshark packet analyzer to display FFShark	Client requires Wireshark to be used.
C2	Must run Wireshark on an external workstation	The ARM machine does not have an environment for GUIs. Thus, Wireshark cannot be run directly on the ARM machine.
C3	Must use the Ethernet port from the ARM machine to connect to the internet	There are a limited number of ports on the FPGA. The current physical setup for the FFShark design has the Ethernet port, connected to the ARM machine, as the only medium for FFShark to send over filtered packets to an external interface [3].
C4	Must not use multiplication, division, and modulo filtering instructions	FFShark omitted the instructions to optimize performance since they are rarely used in filtering applications [3].

Appendix D: Performance Measurement Proof (Author: Alexander Buck)

This appendix provides screenshots of our performance measurements that were used in our report.



Details

File

Name:	/tmp/wireshark_wireshark_extcap_sshdump_20210329024238_3ObhhT_20210329024238_kaPYp1.pcapng
Length:	14 kB
Format:	Wireshark/... - pcapng
Encapsulation:	Ethernet

Time

First packet:	2021-03-29 02:42:38
Last packet:	2021-03-29 02:42:38
Elapsed:	00:00:00

Capture

Hardware:	Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz (with SSE4.2)
OS:	Linux 4.4.0-201-generic
Application:	Dumpcap (Wireshark) 2.6.10 (Git v2.6.10 packaged as 2.6.10-1~ubuntu18.04.0)

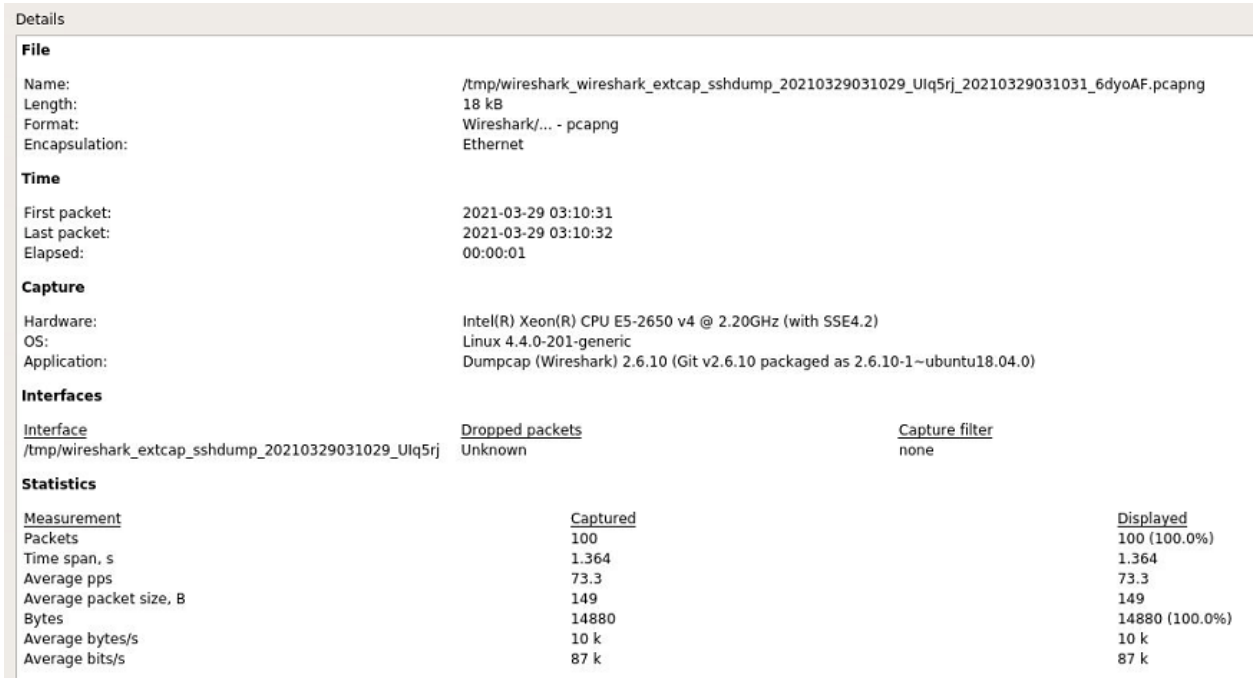
Interfaces

Interface	Dropped packets	Capture filter	Link type
/tmp/wireshark_extcap_sshdump_20210329024238_3ObhhT	Unknown	none	Ethernet

Statistics

Measurement	Captured	Displayed
Packets	100	100 (100.0%)
Time span, s	0.004	0.004
Average pps	27137.1	27137.1
Average packet size, B	107	107
Bytes	10736	10736 (100.0%)
Average bytes/s	2913 k	2913 k
Average bits/s	23 M	23 M

Figure 20: Wireshark output for C code with locking.



Details

File

Name:	/tmp/wireshark_wireshark_extcap_sshdump_20210329031029_UIq5rj_20210329031031_6dyoAF.pcapng
Length:	18 kB
Format:	Wireshark/... - pcapng
Encapsulation:	Ethernet

Time

First packet:	2021-03-29 03:10:31
Last packet:	2021-03-29 03:10:32
Elapsed:	00:00:01

Capture

Hardware:	Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz (with SSE4.2)
OS:	Linux 4.4.0-201-generic
Application:	Dumpcap (Wireshark) 2.6.10 (Git v2.6.10 packaged as 2.6.10-1~ubuntu18.04.0)

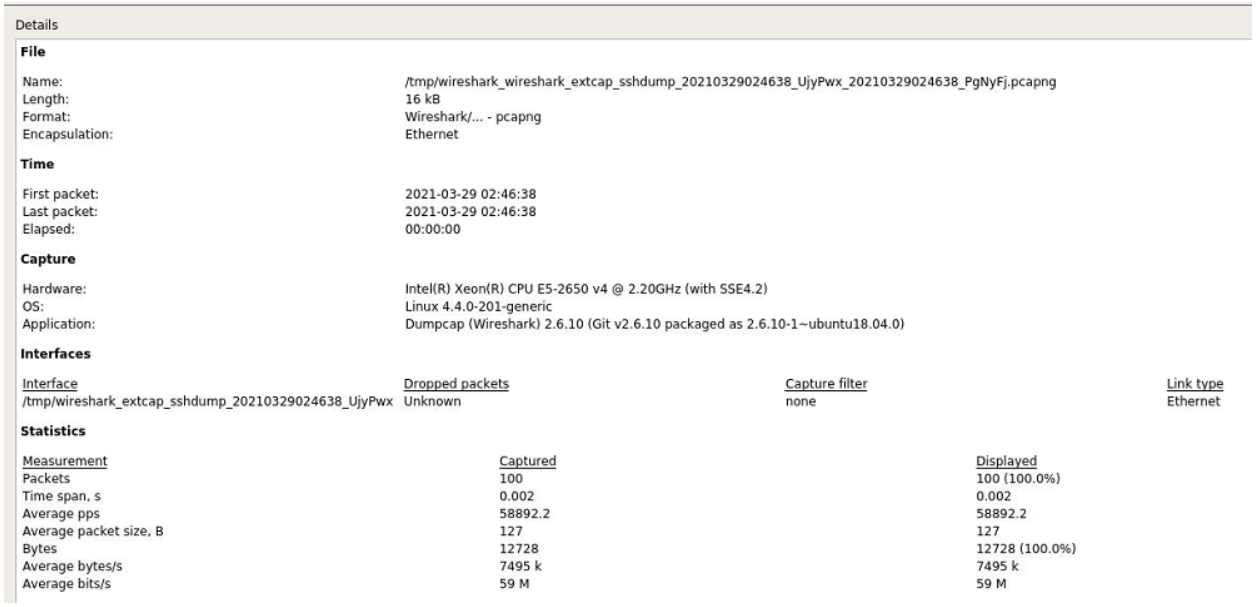
Interfaces

Interface	Dropped packets	Capture filter
/tmp/wireshark_extcap_sshdump_20210329031029_UIq5rj	Unknown	none

Statistics

Measurement	Captured	Displayed
Packets	100	100 (100.0%)
Time span, s	1.364	1.364
Average pps	73.3	73.3
Average packet size, B	149	149
Bytes	14880	14880 (100.0%)
Average bytes/s	10 k	10 k
Average bits/s	87 k	87 k

Figure 21: Wireshark output with Python and locking



Details

File

Name:	/tmp/wireshark_wireshark_extcap_sshdump_20210329024638_UjyPwx_20210329024638_PgNyFj.pcapng
Length:	16 kB
Format:	Wireshark/... - pcapng
Encapsulation:	Ethernet

Time

First packet:	2021-03-29 02:46:38
Last packet:	2021-03-29 02:46:38
Elapsed:	00:00:00

Capture

Hardware:	Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz (with SSE4.2)
OS:	Linux 4.4.0-201-generic
Application:	Dumpcap (Wireshark) 2.6.10 (Git v2.6.10 packaged as 2.6.10-1~ubuntu18.04.0)

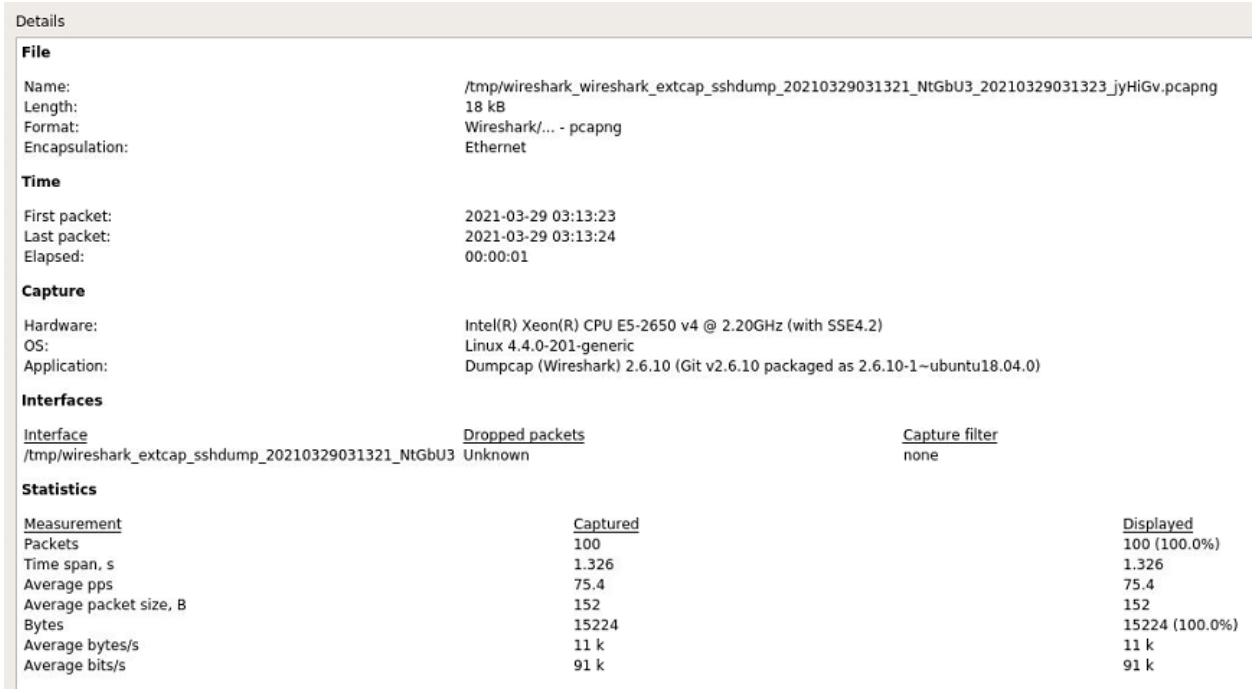
Interfaces

Interface	Dropped packets	Capture filter	Link type
/tmp/wireshark_extcap_sshdump_20210329024638_UjyPwx	Unknown	none	Ethernet

Statistics

Measurement	Captured	Displayed
Packets	100	100 (100.0%)
Time span, s	0.002	0.002
Average pps	58892.2	58892.2
Average packet size, B	127	127
Bytes	12728	12728 (100.0%)
Average bytes/s	7495 k	7495 k
Average bits/s	59 M	59 M

Figure 22: Wireshark output with C code and no locking



Details

File

Name:	/tmp/wireshark_wireshark_extcap_sshdump_20210329031321_NtGbU3_20210329031323_jyHiGv.pcapng
Length:	18 kB
Format:	Wireshark/... - pcapng
Encapsulation:	Ethernet

Time

First packet:	2021-03-29 03:13:23
Last packet:	2021-03-29 03:13:24
Elapsed:	00:00:01

Capture

Hardware:	Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz (with SSE4.2)
OS:	Linux 4.4.0-201-generic
Application:	Dumpcap (Wireshark) 2.6.10 (Git v2.6.10 packaged as 2.6.10-1~ubuntu18.04.0)

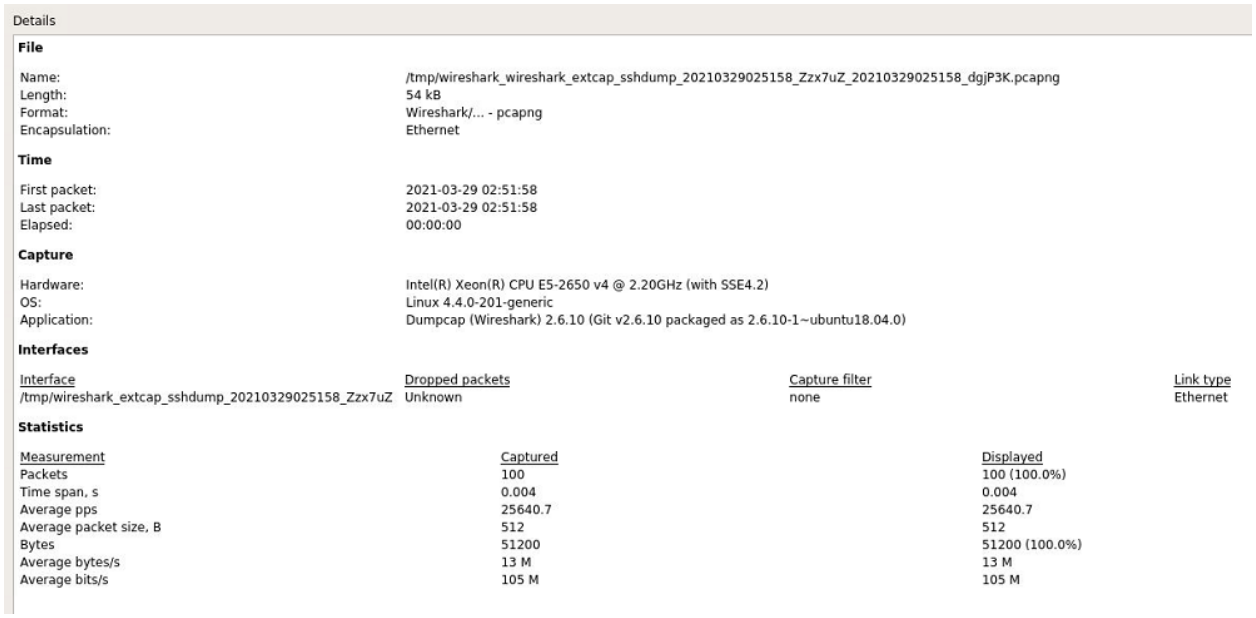
Interfaces

Interface	Dropped packets	Capture filter
/tmp/wireshark_extcap_sshdump_20210329031321_NtGbU3	Unknown	none

Statistics

Measurement	Captured	Displayed
Packets	100	100 (100.0%)
Time span, s	1.326	1.326
Average pps	75.4	75.4
Average packet size, B	152	152
Bytes	15224	15224 (100.0%)
Average bytes/s	11 k	11 k
Average bits/s	91 k	91 k

Figure 23: Wireshark output with Python and no locking



Details

File

Name:	/tmp/wireshark_wireshark_extcap_sshdump_20210329025158_Zzx7uZ_20210329025158_dgjP3K.pcapng
Length:	54 kB
Format:	Wireshark/... - pcapng
Encapsulation:	Ethernet

Time

First packet:	2021-03-29 02:51:58
Last packet:	2021-03-29 02:51:58
Elapsed:	00:00:00

Capture

Hardware:	Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz (with SSE4.2)
OS:	Linux 4.4.0-201-generic
Application:	Dumpcap (Wireshark) 2.6.10 (Git v2.6.10 packaged as 2.6.10-1~ubuntu18.04.0)

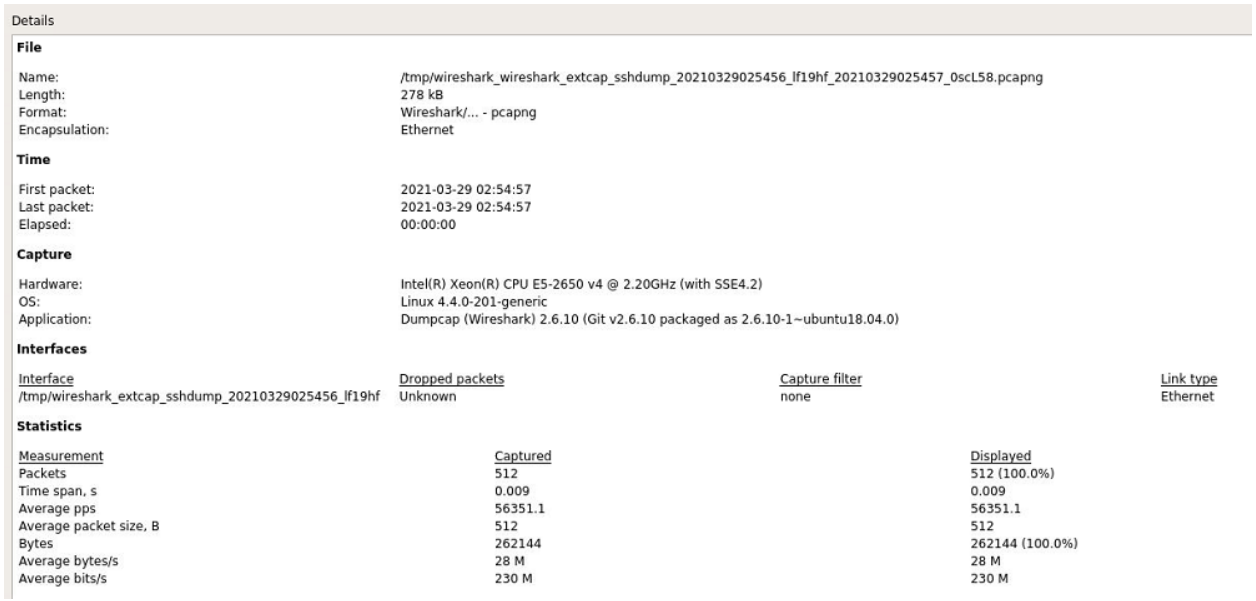
Interfaces

Interface	Dropped packets	Capture filter	Link type
/tmp/wireshark_extcap_sshdump_20210329025158_Zzx7uZ	Unknown	none	Ethernet

Statistics

Measurement	Captured	Displayed
Packets	100	100 (100.0%)
Time span, s	0.004	0.004
Average pps	25640.7	25640.7
Average packet size, B	512	512
Bytes	51200	51200 (100.0%)
Average bytes/s	13 M	13 M
Average bits/s	105 M	105 M

Figure 24: Wireshark output for C code with “perfect” hardware and locking.



Details

File

Name:	/tmp/wireshark_wireshark_extcap_sshdump_20210329025456_lf19hf_20210329025457_0scL58.pcapng
Length:	278 kB
Format:	Wireshark/... - pcapng
Encapsulation:	Ethernet

Time

First packet:	2021-03-29 02:54:57
Last packet:	2021-03-29 02:54:57
Elapsed:	00:00:00

Capture

Hardware:	Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz (with SSE4.2)
OS:	Linux 4.4.0-201-generic
Application:	Dumpcap (Wireshark) 2.6.10 (Git v2.6.10 packaged as 2.6.10-1~ubuntu18.04.0)

Interfaces

Interface	Dropped packets	Capture filter	Link type
/tmp/wireshark_extcap_sshdump_20210329025456_lf19hf	Unknown	none	Ethernet

Statistics

Measurement	Captured	Displayed
Packets	512	512 (100.0%)
Time span, s	0.009	0.009
Average pps	56351.1	56351.1
Average packet size, B	512	512
Bytes	262144	262144 (100.0%)
Average bytes/s	28 M	28 M
Average bits/s	230 M	230 M

Figure 25: Wireshark output for “perfect” hardware and no locking.

```
savi@mpsoc15:~/alex/FFShark_Wireshark_Integration/ffshark_c_lang_drivers$ ./ffshark_only_read -n 100

Total time : 0.001351 seconds
Data rate : 94792005.921540 bits/second
```

Figure 26: Terminal output of reading from FFShark without doing any data transfer to Wireshark.

Details

File

Name:	/tmp/wireshark_wireshark_extcap_sshdump_20210329032844_6PqXEg_20210329032844_I3YuSw.pcapng
Length:	199 MB
Format:	Wireshark/... - pcapng
Encapsulation:	Ethernet

Time

First packet:	2021-03-29 03:28:44
Last packet:	2021-03-29 03:28:48
Elapsed:	00:00:03

Capture

Hardware:	Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz (with SSE4.2)
OS:	Linux 4.4.0-201-generic
Application:	Dumpcap (Wireshark) 2.6.10 (Git v2.6.10 packaged as 2.6.10-1~ubuntu18.04.0)

Interfaces

Interface	Dropped packets	Capture filter	Link type
/tmp/wireshark_extcap_sshdump_20210329032844_6PqXEg	Unknown	none	Ethernet

Statistics

Measurement	Captured	Displayed
Packets	145826	145826 (100.0%)
Time span, s	3.357	3.357
Average pps	43445.0	43445.0
Average packet size, B	1337	1337
Bytes	194973804	194973804 (100.0%)
Average bytes/s	58 M	58 M
Average bits/s	464 M	464 M

Figure 27: Wireshark output when transferring a large file through ssh protocol.