# FFShark: A 100G FPGA Implementation of BPF Filtering for Wireshark

Juan Camilo Vega
Department of Electrical and Computer Engineering
University of Toronto
camilo.vega@mail.utoronto.ca

Marco Antonio Merlini
Department of Electrical and Computer Engineering
University of Toronto
marco.merlini@utoronto.ca

Paul Chow
Department of Electrical and Computer Engineering
University of Toronto
pc@eecg.toronto.edu

***Abstract*—Wireshark-based debugging can be performed on ordinary desktop computers at 1G speeds, but only powerful computers can keep up with 10G. At 100G, this debugging becomes virtually impossible to perform on a single machine.**

**This work presents FFShark, a Fast FPGA implementation of Wireshark. The result is a compact, relatively inexpensive passthrough device that can be inserted into any running 100G network. Packets will travel through FFShark with no interruption and minimal additional latency. A developer can send standard Wireshark filter programs to the FFShark device at any time; packets that satisfy the filter will be copied and sent back to the developer's workstation over a separate connection.**

**We show that our open source passthrough device has lower latency than commercial 100G switches, and that our design is already capable of handling 400G speeds.**

## I. Introduction

Wireshark [1] is a software tool that allows network developers and administrators to inspect live network packets without disrupting communications. This inspection requires that a specified subset of packets be captured and then relevant fields of the packet can be analyzed. This capability is invaluable for network analysis and debugging. A key benefit of Wireshark is its usage of the BSD Packet Filter (BPF), which is supported by most OS kernels. The BPF technique reduces memory copying and leads to significant performance improvements.

High-performance processors can perform packet filtering at 10G. As we move to 100G and beyond, using Wireshark becomes impossible. For example, a 9th generation Intel i9-9900KS processor is equipped with sixteen PCIe3.0 [2] lanes, each with a maximum bandwidth of 8 Gbps [3]. To communicate 100 Gbps from the NIC to the CPU would require the bandwidth of 13 out of 16 available PCIe lanes, assuming no overhead. Beyond that, given a clock speed of 5 GHz [2], the CPU would have to receive a packet via PCIe, filter it based on the user's specification, and potentially copy out the packet in 1.6 or fewer clock cycles per 32 bit word. Even if the CPU is otherwise unloaded and never suffers a cache miss, it is still impossible to perform the requisite filtering at these speeds. Aggressive multithreading might begin to make 100G filtering possible on a CPU under very optimistic assumptions, but it is impractical and not scalable to faster speeds.

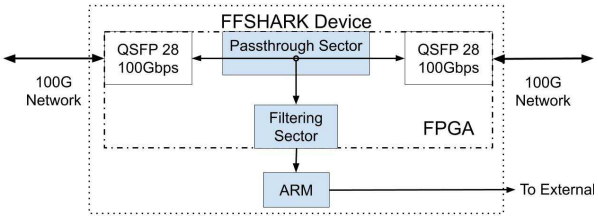


Fig. 1. Architecture of FFShark. The Passthrough Sector is detailed in Fig. 4 and the Filtering Sector is detailed in Fig. 5.

To use Wireshark beyond 10 Gbps speeds, we propose FFShark. FFShark is an open source [4], low-latency passthrough device that can be placed between any two points in a network. It supports arbitrary filters written in the PCAP filter syntax[1] [5]. In debug, this can be used to perform network-wide analysis of 100G traffic, *e.g.*, monitoring all traffic between two switches or between a data center and the wide area network. The additional latency cost of FFShark is comparable to that of a switch, making live system testing possible.

FFShark is implemented using FPGA technology, where a parallel array of filters will snoop on packet data (Fig. 1). The filters are implemented as CPUs within the FPGA fabric and natively emulate the BSD Packet Filter virtual machine [6]. Additional circuitry distributes the incoming high-speed network line into multiple lower speed streams, one for each filtering CPU. FFShark can currently perform Wireshark filtering in a real 100G network, but we will show that it can operate correctly at speeds up to 400G (Section IV-C) and can be immediately used once transceivers become available.

The remainder of the paper is organized as follows: Section II offers a background discussion on Wireshark and the BSD Packet Filter method. Section III presents related work. Section IV details the design of FFShark, including high-speed techniques and the filtering CPU. Section V presents the implementation results. Finally, Section VI discusses future work and Section VII concludes the paper.

[1]This is the syntax used by Wireshark

2576-2621/20/$31.00 ©2020 IEEE
DOI 10.1109/FCCM48280.2020.00016

## II. Background

On a typical desktop computer all incoming packets are processed by the OS kernel and copied into the memory of the correct user application. For live network debugging, a user application such as Wireshark requests that packets are *also* copied into its own memory. This section explains the method used by Wireshark to efficiently copy packets of interest; this method relies on the BSD Packet Filtering technique, which is also explained in this section.

### A. Wireshark Architecture

Wireshark allows a developer to request copies of packets from the OS. Furthermore, the developer may not wish to see all packets, so Wireshark also allows unwanted packets to be hidden. A naïve implementation would be to copy every single packet into Wireshark's memory, whereupon Wireshark would only display the packets of interest. A much more efficient solution would be to avoid copying unwanted packets in the first place.

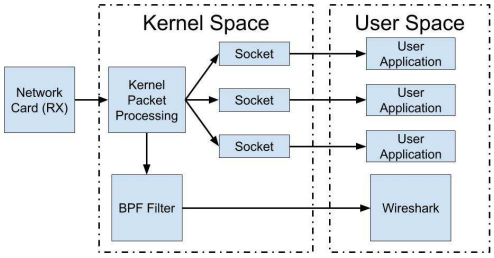


Fig. 2. Standard Wireshark operation.

Fig. 2 illustrates how Wireshark operates in a standard OS environment. The developer enters a filter specification in PCAP syntax [5]. For example, the expression `tcp src port 100` selects only TCP packets originating from port 100. Wireshark compiles this expression into BPF machine code (described in the next subsection) and installs the filter code using a kernel system call. Packets arriving at the Network Card are directed by the Kernel Packet Processing through Sockets to the relevant User Application(s). Additionally, all packets are copied to the BPF Filter and any packets that match the filter described by the filter code are copied to Wireshark's user memory.

### B. The BSD Packet Filter (BPF)

The BPF method results from the observation that "filtering packets early and in-place pays off" [6]. Using this approach, a user submits a BPF program that the kernel will execute on each incoming packet. These programs are a sequence of machine code instructions that are executed by an emulator in the OS kernel. Any Wireshark-compatible OS is responsible for correctly emulating the BPF machine. The kernel will only copy the packet back to the user depending on the BPF program's return value.

A brief summary of the BPF machine is as follows. There are two 32-bit registers: the Accumulator (`A`) and the auXiliary (`X`). The processor has byte-addressable read-only access to an entire packet (headers included) and read/write access to a small scratch memory. An instruction is defined by its class (Table I), addressing mode, jump offsets, and immediate value. The BPF instruction layout is shown in Fig. 3.

TABLE I
BPF Instruction Classes

| Name | Description |
|---|---|
| `LD` | Load from memory into `A` register |
| `LDX` | Load from memory into `X` register |
| `ST` | Store `A` register to scratch memory |
| `STX` | Store `X` register to scratch memory |
| `ALU` | Arithmetic and Logic instructions |
| `JMP` | Conditional and unconditional branching |
| `RET` | Returning (to signify acceptance or rejection) |
| `MISC` | Miscellaneous |

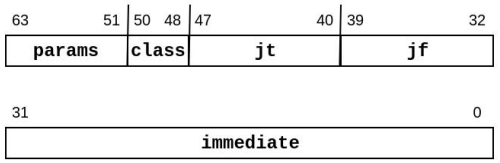


Fig. 3. BPF instruction layout. The numbers are bit indices.

## III. Related Work

Campbell and Lee [7] implemented a 100G Intrusion Detection System (IDS) using only commodity hardware. A 100G router was used to evenly distribute packets to several worker machines [8]. To further reduce the load on individual machines, a central management node can allow certain types of traffic to bypass the IDS once determined to be safe. The architecture of this IDS requires the operation of a 100G load balancer and several high-performance CPU machines. This has the advantage of using immediately available parts but is not cost-effective.

A number of commercial 100G NICs and switches exist that implement support for hardware packet filtering, each one presenting its own proprietary API. Simple PCAP expressions can be converted by nBPF [9] [10] to these vendor-specific formats, but depending on a specific NIC's supported operations only a limited subset of expressions can be translated. Where nBPF makes use of filtering capabilities of commercial 100G hardware, FFShark natively implements the full BPF standard as a separate device that can be inserted into a network at any location. This satisfies our goal of maintaining full compatibility with Wireshark.

Another type of 100G packet filtering involves automatically generating an FPGA design from a higher-level filter description. This method takes advantage of the high performance and reconfigurability of FPGAs, but aims to meet the needs of system administrators who would prefer to use simple packet filtering specifications. Xilinx netCope [11] generates VHDL from a P4 filter specification [12]. Additionally, the more general technique of High-Level Synthesis can potentially allow a developer to write their own packet filtering algorithm and implement it on an FPGA. These approaches offer little to

no support for changing the filter specification interactively; FFshark is an FPGA overlay that conforms to the standard BPF interface and does not require generating a new FPGA configuration when changing filters.

FMAD Engineering produces a sustained 100G packet capture solution [13]. The product is a rack-mounted box with two QSFP28 input ports and an array of ten SSDs; the device accepts BPF filters and saves accepted packets for later review. FMAD is a commercial product that supports the same filtering capability as FFShark and can run at speeds up to 100G. However, FFShark is open source and is available for the community to customize and use for experimentation. FFShark is also shown to be usable in a 400G network.

Bittware produces a closed source 10/25/40/100G packet broker device with packet filtering [14]. This is a PCIe expansion card with four QSFP28 ports. For filtering it supports 10G speeds and runtime configuration of filter parameters, which can be synthesized from PCAP filter expressions. By changing the FPGA image, the solution can be upgraded to support 100G filtering without needing additional hardware. The description of this Bittware product says that it supports a "set of industry-standard PCAP ASCII expressions" implying that it does not have the full flexibility of the BPF engine, such as in FFShark. Also, like the FMAD product just described, the Bittware product is a commercial product while FFShark has the advantages of being open source and being ready for 400G speeds.

## IV. Design

Fig. 1 shows an overview of FFShark. The design consists of a Passthrough Sector (Fig. 4) and a Filtering Sector (Fig. 5). The FPGA and ARM components are implemented using a Xilinx Zynq Ultrascale+ XCZU19EG-FFVC1760-2-I (MPSoC) [15]. The Filtering Sector is itself divided into three subcomponents: a Chopper, a number of BPF Cores, and a Forwarder. The Chopper splits the high-speed input data into several queues operating at a slower speed. Each queue feeds into a BPF Core, which executes a BPF filter program. Finally, if a packet is accepted, the Forwarder sends it out of the filter. Each of these subcomponents is detailed in a separate subsection, below.

### A. Passthrough Sector

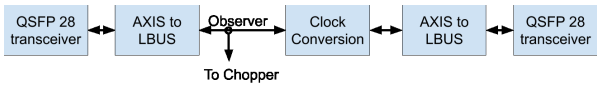


Fig. 4. Design of the passthrough sector.

The Passthrough Sector is shown in Fig. 4. The 100G hardened CMAC in the Ultrascale+ device provides a local bus interface called LBUS. Since all other Xilinx cores use AXI, an LBUS-to-AXIS converter circuit [16] converts the raw signals from the PHY layer to standard AXI Stream messages and vice-versa. The AXI Streaming channels are connected together allowing all messages to be directly forwarded to the opposite port.

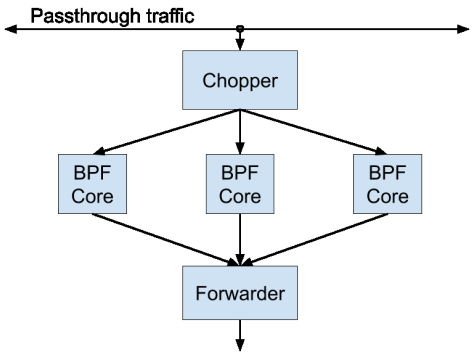


Fig. 5. Design of the Filtering Sector.

The two QSFP28 transceivers operate at a clock speed of approximately 323 MHz, but are driven by two separate clocks. Because of this, additional logic is needed to convert messages from one clock domain to the other. Despite using slightly different clocks, both transceivers are rated for operation at 100G and can sustain this data rate given that there are gaps between bursts of traffic. FIFO buffering was added to allow for these random short-term bursts.

The passthrough traffic is sent over an AXI Stream channel. The wires constituting this channel also feed directly into the Chopper, allowing it to observe any passing communication.

### B. Chopper

The Filtering Sector is designed to allow a user to specify any filter (i.e. BPF program). Since the number of instructions (and thus, the length of the program's runtime) is not known *a priori*, we allow the user to distribute the task among multiple parallel BPF Cores as shown in Fig. 5. For this, we implemented a Chopper with one output queue per packet filter. This also allows us to clock the BPF CPUs at a slower frequency, which significantly eases the burden on the FPGA compilation for complex structures [17].

This architecture allows us a significant margin for upward scalability. The BPF Cores have a maximum operable bit rate; even so, when moving to 400G speeds, the Chopper can be easily reconfigured to divide the input bit rate among a larger number of slower output queues.

The Chopper is implemented in HLS as follows. A single AXI Stream input receives the packets. This input can be clocked up to 475 MHz, and there is no limitation on the data width of the input channel. An arbiter, based on probes that detect forward congestion and buffer usage, and based on its recent history of where it sent the previous few packets, selects an output stream for each packet to maintain the average bit rate of each output line under the specified amount. The decision logic is separate from the crossbar allowing the chopper to have a smaller critical path. The output decision is based on four-cycle-old information. However, since we have sufficient output buffering to fit at least one packet plus four flits, this does not compromise reliability. Packets are considered indivisible; an entire packet from the input stream is sent to the same output stream, where the data is

buffered in a FIFO memory. A second circuit reads from this FIFO memory and outputs it as an AXI Stream signal of the appropriate clock speed and channel width to support the BPF Core's maximum bit rate.

The arbitration portion of the Chopper sets the average bit rate on each output line to 50 Gbps. The indivisibility of the packets, however, require that the Chopper accomplish this using intermittent bursts of 165 Gbps (512 bit wide signal at 322MHz). It is for this reason that the data needs to be buffered, as the BPF Cores cannot support any bursts beyond 51.2 Gbps. To prevent this FIFO buffer from overflowing, the capacity is set to be large enough to buffer two maximum-sized packets (3 kB, or 18 kB with jumbo packets enabled). Even if a packet has not been fully read, the buffer can still accept an entire new packet. Testing with a random assortment of packets ranging from 64 B to 9 kB in size showed that no packets were ever dropped when the average input bit rate was kept at 100 Gbps.

The arbiter is able to function regardless of the composition of the network traffic (*i.e.*, distribution of small and large packets). However, the parallel approach can lead to packets being recorded out of order. To resolve this, a timestamp (based on a global FPGA clock counter) is recorded with each incoming packet and added to its header. Currently, software can perform the task of re-ordering the packets, however, it is more desirable to do the re-ordering in hardware. One option is to remove timestamps altogether and instead modify the Chopper and Forwarder to never output packets out of order; this would have the lowest additional FPGA resource cost, and would only suffer a small loss of maximum throughput. Alternatively, a general re-order buffer would preserve maximum throughput at the cost of significantly larger on-chip memory costs. These developments are left as future work.

### C. Clocking

The Passthrough Sector uses one 512-bit AXI Stream channel clocked at 322 MHz in each direction. Each channel supports a maximum bit rate of 164 Gbps (512 b at 322 MHz). The Chopper snoops on this channel at full speed and outputs several streams operating at a clock speed of 100 MHz with a bus width of 512 bits for a bit rate of 51.2 Gbps, which is an acceptable speed for the BPF Cores. Each BPF Core is capable of outputting accepted packets at this same rate of 51.2 Gbps.

Currently, we have no available 400G hardware. However, we performed a 400G Chopper test with an internally generated signal of random packets between 40 and 9000 bytes in size, clocked at 450MHz. The bus width was 1024 bits wide for a maximum bitrate of 460.8G. Even at this bitrate, the Chopper was able to correctly divide the input into nine output signals at 51.2 Gbps (512-bit bus at 100 MHz) with no packet drops.

### D. BPF Cores

Currently, Wireshark relies on the OS kernel to receive packets from networking hardware and perform filtering based on an arbitrary BPF program. As established in Section I, even a powerful CPU machine cannot perform either one of these tasks at 100G. FFShark moves the filtering operation from the OS kernel to a passthrough device embedded within the 100G network, and performs filtering with an array of parallel processors we call *BPF Cores* (Fig. 5).

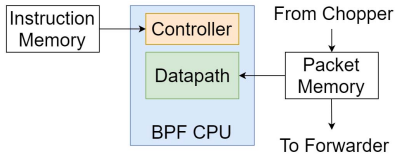


Fig. 6. BPF Core.

Each BPF Core is equipped with its own Instruction Memory and Packet Memory, as shown in Fig. 6. Every packet from the Chopper is copied into the Packet Memory of a BPF Core. Each core's Instruction Memory can, at runtime, be loaded via an external configuration bus with a BPF program that will be executed once for each packet it receives. Using `RET` instructions, a program causes the BPF Core to emit an accept or reject signal; on acceptance, the Forwarder (Section IV-G) will send the packet to external storage.

The BPF CPU consists of a Datapath and a Controller. The Datapath is shown in detail in Fig. 7. The Controller is pipelined with three stages: fetch, execute, and writeback. The last two stages control the Datapath via the Control Signal wires, and support all BPF operations except `MUL`, `DIV`, and `MOD` (which are seldom-used in real filtering applications). There are four Control Signals that do not connect into the Datapath: the `inst_rd_en` and `pack_rd_en` signals are directly connected to their respective memories, and `cpu_acc` and `cpu_rej` are used to signal the Ping-Pang-Pong interconnect as described in Section IV-F. The Datapath is connected to the Instruction Memory via its `inst_rd_data` wire, and is connected to the Packet Memory via its `pack_rd_addr` and `pack_rd_data` wires. All other wires in Fig. 7 are internal and do not leave the module.

### E. Selecting the Number of Parallel BPF Cores

The Chopper must split the input stream among a number of parallel BPF Cores, which have a maximum operable bit rate. However, if the BPF programs are very long and/or the BPF Cores are swamped with an excess of small packets, the effective bit rate of each core is decreased and more are required to support the 100G bandwidth.

This subsection will present a general technique for estimating the required number of parallel cores along with a running example, shown in Listing 1. This listing shows the BPF instructions resulting from the `tcp src port 100` PCAP filter expression. For our running example, we will assume that 20% of input packets use neither IPv4 nor IPv6, 30% of packets use IPV4 and UDP, 25% of packets use IPv4 and TCP but are not from source port 100, and 25% of packets use IPv4 and TCP and are from source port 100. For our example, the average packet length will be 680 B.

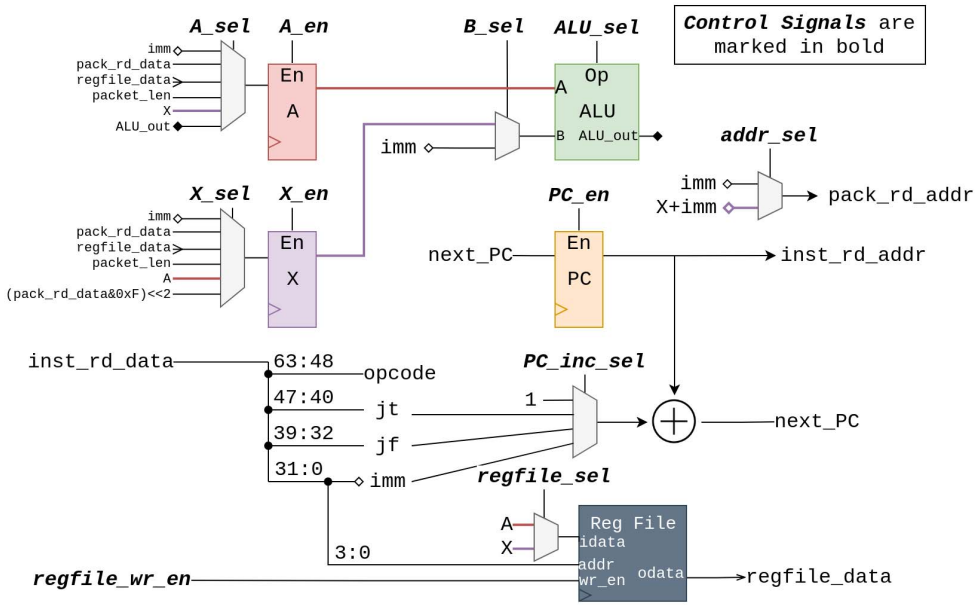


Fig. 7. BPF CPU Datapath.

```
        ldh [12]                ; A = EtherType
        jeq #0x86dd, ipv6, ipv4 ; Check if IPv6

ipv6:   ldb [20]                ; A = Protocol Number
        jeq #0x06, srcv6, rej   ; Check if TCP
srcv6:  ldh [54]                ; A = TCP source port
        jeq #100 acc, rej       ; Check if 100

ipv4:   jeq #0x0800 tcp4, rej   ; Check if IPv4
tcp4:   ldb [23]                ; A = Protocol Number
        jeq #0x06 frag0, rej    ; Check if TCP
frag0:  ldh [20]                ; A = Fragment #
        jset #0x1fff rej, srcv4 ; Check Fragment # != 0
srcv4:  ldxb 4*([14]&0xf)       ; X = IP Header Length
        ldh [x+14]              ; A = TCP source port
        jeq #100 acc, rej       ; Check if 100

acc:    ret #0xffff             ; Accept
rej:    ret #0                  ; Reject
```

Listing 1. Compiled BPF machine code for the PCAP filter `tcp src port 100`

In general, the input stream of packets can be broken up into $k$ different classes, where each class triggers an identical execution path in the filter code. In our running example, $k = 4$. Each packet in class $i$ has a code path of length $I_i$ instructions and has an average packet length of $l_i$ bits. Finally, a randomly selected input packet has a probability $P_i$ of belonging to class $i$. The values of $I_i$, $l_i$, and $P_i$ for the running example are shown in Table II.

TABLE II
$I_i$, $l_i$, AND $P_i$ VALUES FOR RUNNING EXAMPLE IN SECTION IV-E

| Class ($i$) | $I_i$ | $l_i$ | $P_i$ |
|---|---|---|---|
| 1 | 4 | 5440 | 20% |
| 2 | 6 | 5440 | 30% |
| 3 | 11 | 5440 | 25% |
| 4 | 11 | 5440 | 25% |

For these calculations we will assume the BPF CPU requires four cycles per instruction[2] *i.e.*, CPI $= 4$. If the CPU is operating with a clock period of $T$ seconds, then the average bit rate of the BPF CPU ($R_{\text{CPU}}$) is

$$R_{\text{CPU}} = \sum_{i=1}^{k} P_i \times \frac{l_i}{I_i \cdot \text{CPI} \cdot T} \tag{1}$$

The denominator in Equation 1 represents the amount of time (in seconds) needed by the CPU to process a single packet.

The quantity $R_{\text{CPU}}$ represents the bit rate of a BPF CPU. However, to select the correct number of BPF *Cores*, we must compute $R_{\text{Core}}$, the bit rate of a BPF Core:

$$R_{\text{Core}} = \min\left(R_{\text{CPU}}, 512 \text{ bits} \times 100 \text{ MHz}\right) \tag{2}$$

where $512 \text{ bits} \times 100 \text{ MHz}$ represents the maximum bit rate that can be used to fill the BPF Core's Packet Memory (c.f. Section IV-C).

Thus, the required number of parallel BPF Cores is $N = \lceil 100\text{G}/R_{\text{Core}} \rceil$. In our example $R_{CPU} = 19.8\text{G}$, therefore $R_{Core} = 19.8\text{G}$ and $N = 6$. In a real-world application, an FFShark user would have to make educated guesses of $k$, $I_i$, $l_i$, and $P_i$ based on their knowledge of the system they are debugging.

Generally speaking, we have found six parallel cores to be sufficient for common filters such as selecting all packets to/from a specific address or port range. More complex filters may require more BPF Cores, and the described procedure for estimating the number needed should be done. However, even as a filter becomes more complex, filter code is able

[2]This is an upper bound. For common Wireshark filters, the average CPI is closer to 2.5

to terminate early on rejected packets and the proportion of accepted packets usually decreases. Consider the example shown in Table III: this example models a filter with more conditions applied to the same traffic pattern as our previous example. Specifically, the filter checks for any HTTP packet containing data. Classes 1 and 2 are still rejected early, and class 4 requires two additional instructions before getting rejected. Class 3, however, splits into two classes: 3a are packets that pass some of the conditions of the new filter but are ultimately rejected, 3b are packets that pass all conditions. Even if the number of instructions for accepted packets is much larger, in this example the number of required parallel cores is still six.

TABLE III
$I_i, l_i$, AND $P_i$ VALUES FOR THE FILTER `TCP PORT 80 AND (((IP[2:2]-((IP[0]&0XF)<<2))- ((TCP[12]&0XF0)>>2)) != 0)`

| Class | $i$ | $I_i$ | $l_i$ | $P_i$ |
|---|---|---|---|---|
| 1 | 1 | 4 | 5440 | 20% |
| 2 | 2 | 6 | 5440 | 30% |
| 3a | 3 | 13 | 5440 | 20% |
| 3b | 4 | 28 | 5440 | 5% |
| 4 | 5 | 13 | 5440 | 25% |

### F. Ping-Pang-Pong Controller

The BPF Core is a BPF CPU equipped with an Instruction Memory and a Packet Memory (Fig. 6). Immediately we notice that the Packet Memory is shared among three agents: the Chopper, the BPF CPU, and the Forwarder. An important performance optimization is a "ping-pang-pong" buffer, shown in Fig. 8. This scheme enables all three agents to access memory at full speed simultaneously.

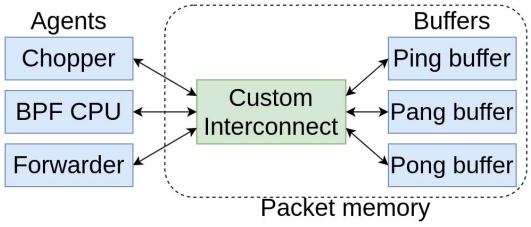


Fig. 8. Ping-pang-pong buffer.

The custom interconnect maintains three FIFOs that represent a job queue for each agent. At any given moment, the *token* at the head of an agent's queue is used by a multiplexer to connect the agent with one of the Ping, Pang, or Pong buffers.

Fig. 9 demonstrates the technique in action. The top-left block represents the initial state of the job queues and is understood to mean "all three buffers are waiting for the Chopper to read a packet, and the Chopper is currently connected to the Ping buffer". When the Chopper has finished reading a packet into the Ping buffer, the Ping token is popped from the Chopper's queue and added to the CPU's queue (top-right block). At this point, the Chopper is now connected to the Pang buffer and the CPU begins processing the packet in the Ping buffer. The bottom-left block shows what would happen if the Chopper finished reading a packet before the CPU finished executing its program: the Pang token is popped from the Chopper's queue and added to the CPU's queue. The bottom-right figure shows the result of the CPU accepting the packet: the Ping token is popped from the CPU's queue and added to the Forwarder's queue.

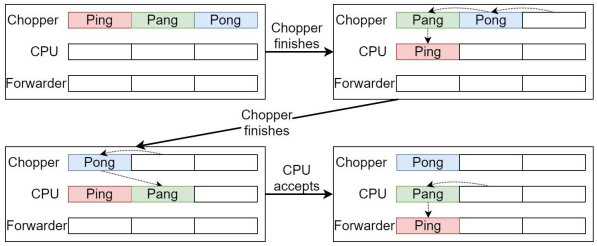


Fig. 9. Job queues in the Ping-Pang-Pong interconnect.

### G. Forwarder

Upon acceptance of a packet, the BPF Core places one of the Ping, Pang, or Pong buffers on the Forwarder's job queue. The Forwarder then reads from the buffer and outputs packet data on a 512-bit wide AXI Stream output at 100 MHz.

If the target FPGA device is equipped with three (or more) QSFP28 ports, the accepted packet streams can be recombined and sent out at the full 100G rate. The FPGA used in this work has only two QSFP28 ports, but has a high-bandwidth channel to a connected ARM CPU.

Borrowing from the work of Vega et al. [18], we extended their protocol to transfer bidirectional messages between the FPGA and the on-board ARM processor. Using this scheme, we were able to concatenate the accepted packets from the BPF Cores and forward these to the ARM core. A header was also added to allow some metadata to be transferred with the packet. Once in the ARM core, the message can be handled as desired. For functionality purposes, initial tests printed the filtered messages onto the terminal of the ARM processor. For a more practical implementation, a separately developed storage system [19] allowed us to reliably, and with low overhead store the collected messages to a remote storage server. This system can be used by both FPGAs and CPUs. However, since we lack a third network port, packets are forwarded to the ARM to use the ARM's network port.

To utilize this channel for the stream of accepted packets, we built a custom data width converter for the Forwarder that keeps the clock at 100 MHz but reduces the bus width to 64 bits for a bitrate of 6.4 Gbps. This is slow enough for the ARM processor to receive the accepted packets. If the bandwidth of *accepted* packets exceeds the speed of this 6.4 Gbps bottleneck, the Filtering Sector will drop packets, but the Passthrough Sector is not affected.

## V. RESULTS AND DISCUSSION

For these results, the FFShark system was implemented on a Zynq Ultrascale+ XCZU19EG-FFVC1760-2-I [15]. This

chip includes both an FPGA and an ARM CPU on the same silicon die. The two can communicate via several high-speed AXI channels. The FPGA has 1.1 million logic elements and 9.8 MB of on-chip memory. The processor is a 64-bit ARM Cortex-A53. This chip is part of a Fidus Sidewinder 100 [15] FPGA board, which connects the MPSoC to two QSFP28 4x25G ports, a 1G Ethernet port, and a host of other peripherals.

To characterize the performance of FFShark, three types of measurements are gathered: the added latency on passthrough traffic, the packet drop rate and performance of the Filter and Passthrough Sectors, and the number of FPGA resources needed by the design.

### A. Insertion Latency

The goal of this test is to measure $l_P$, the insertion latency of the passthrough sector of FFShark, defined by the time taken for a packet to enter on one QSFP port and leave on the other. Fig. 10 outlines the test setup. Each gray box in the figure represents a Sidewinder board, and each connection is a 2 m 100G cable. UDP was chosen as a lightweight routable protocol with consistent latency that can be subtracted from the total latency.

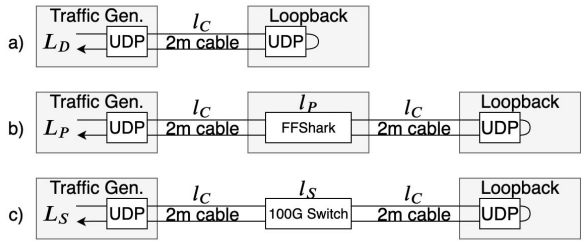


Fig. 10. Test setup for latency measurements

The round-trip time of a direct connection was measured, as shown in Fig. 10a. This quantity is denoted by $L_D$, and represents the time taken for a packet to:

1) Undergo processing by a 100G UDP core [16]
2) Pass through 2 m of cable ($l_C$)
3) Enter and exit the Loopback
4) Pass through 2 m of cable
5) Upon return to the Traffic Generator, undergo processing by a 100G UDP core.

The Traffic Generator board saves a timestamp when a packet is sent to the UDP core (Step 1) and when the packet exits the UDP core after returning (Step 5).

Next, the round-trip time including FFShark was measured as shown in Fig. 10b. This quantity is denoted by $L_P$.

TABLE IV
ROUND-TRIP TIMES FOR DIRECT CONNECTION AND PASSTHROUGH TEST

| Size (B) | 100 | 164 | 292 | 420 | 548 | 804 | 1060 | 1572 |
|---|---|---|---|---|---|---|---|---|
| $L_D$ ($\mu$s) | 3.72 | 3.04 | .893 | .914 | .952 | 1.01 | 1.08 | 1.21 |
| $L_P$ ($\mu$s) | 5.78 | 3.66 | 1.64 | 1.66 | 1.72 | 1.82 | 1.93 | 2.14 |

Table IV presents the measured values of $L_D$ and $L_P$ for various packet sizes (including the 36 byte UDP header), averaged over 4000 packets in a continuous 100G transmission. Given the propagation speed through a copper cable, the propagation delay $l_C$ of a 2 m cable was found to be 0.009 $\mu$s [20]. Fig. 11 shows the calculated value of $l_P = (L_P - L_D - 2 \cdot l_C)/2$. A similar test was performed with a Dell Z9100-ON 100G switch in lieu of FFShark, as shown in Fig. 10c. The measured insertion latency of this switch $l_S$ is also presented in Fig. 11 to present a comparison between FFShark and commodity 100G hardware. Note that the switch does not support packets larger than the MTU limit of 1.5KB.

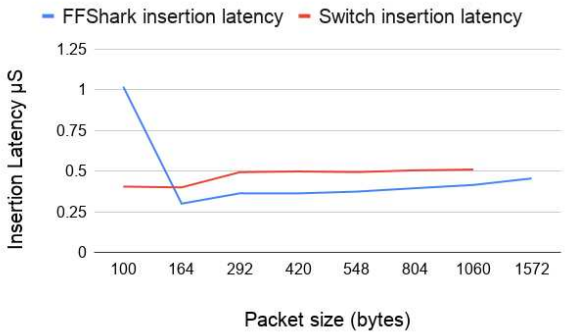


Fig. 11. Insertion latency of FFShark ($L_P$) vs. a Dell Z9100-ON 100G switch ($L_S$).

A higher latency was recorded for packets smaller than 100 bytes. This was due to our test exceeding the maximum packet rate for small packets as explained in SectionV-D. When the bit rate was lowered, the insertion latency for packets 100 bytes in size or smaller was approximately equal to the latency of the test for packets 164 bytes in size (approximately 0.3 $\mu s$). Additionally, the latency of FFShark increases as the packet size increases. At this time, we are using Xilinx CMAC controllers, which force us to store-and-forward entire packets instead of using a cut-through approach[3].

### B. Maximum Performance

To test maximum performance, 32768 packets ranging in size from 16 B to 1500 B were sent through FFShark at the full 100G bitrate to a loopback device and were re-collected at the origin. The device at the origin verified the integrity of the packets and counted the number of packets lost or corrupted. 99.41% of packets were returned correctly in this highly congested application. No packets were returned corrupted. When the test was modified to test packets ranging in size from 176B to 1500B, no packets were dropped. Section V-D explores the source of these packet drops. Handshaking, or flow control can be added to mitigate these packet drops.

The Passthrough Sector and Filtering Sector were shown to never miss a packet[4]. This was demonstrated by running a test multiple times where a single packet with a unique header was hidden in different locations within a 100G burst of traffic. The packet filter was programmed to detect the signature. In

[3]The Xilinx CMAC does not tolerate "bubbles" in its input data stream

[4]Packet drops may occur if the rate of accepted packets exceeds the bitrate allowed by the Forwarder for an extended period of time

TABLE V
RESOURCE USAGE OF FFSHARK AND SELECTED SUBCOMPONENTS

| Component | LUTs | FFs | BRAMs |
|---|---|---|---|
| Single BPF Core | 3,600 | 1,161 | 23.5 |
| Chopper | 4,112 | 8,122 | 0 |
| Entire FFShark Design | 56,250 | 72,509 | 251 |
| Available in Zynq Ultrascale+ MPSoC | 522,720 | 1,045,440 | 986 |

all cases FFShark was able to correctly identify and forward the packet to the ARM, while ignoring all other packets.

### C. Resource Usage

Table V shows the resource usage of this project. The first two rows show the resource usage of a single BPF Core and a Chopper. The third row shows the entire cost of the final FFShark design; this tally includes six BPF cores, a Chopper, and the extra logic required for the Passthrough Sector and for interfacing the Instruction Memories and Forwarder with the on-board ARM CPU. No DSPs were used in this design. Since this design only uses a fraction of the FPGA's total capacity, the remaining resources can be used to add more filtering cores, or to embed FFShark into a larger design. Alternatively, FFShark could be placed on a smaller, less expensive FPGA.

### D. Discussion

High latency and occasional packet drops were seen in tests involving packets smaller than 176B at the 100G bit rate. It was discovered that the 100G CMAC and transceiver on the FPGA has per-packet overhead [21] creating a maximum packet rate constraint in addition to the maximum bit rate constraint. In tests with small packets, the number of packets per second exceeded this maximum and the network hardware began applying backpressure. The GULF Stream UDP core has internal buffers that temporarily mitigate this issue. However, as these buffers fill up, the backlog of packets means that latency increases linearly leading to high latency at small packet sizes as seen in Section V-A. Eventually, the GULF Stream FIFOs become completely full resulting in the few congestion-based packet drops seen in Section V-B. When the bit rate was decreased for the smaller packet sizes, the packet drops ceased and the latency followed the same trend for all packet sizes.

Apart from this hardware-based restriction, FFShark performs as expected adding a smaller latency than a high-end 100G switch and without affecting packet data. The filters are able to inspect 100% of incoming packets when the Forwarder's bandwidth is not exceeded.

The total design occupies less than a quarter of the resources in the FPGA and the Chopper is able to correctly split the input stream at 400G speeds. This makes FFShark immediately scalable to 400G when MAC/transceiver support becomes available.

## VI. FUTURE WORK

This work has many future prospects as it is scalable to a wide range of speeds and is very versatile. While in this case it is used to interactively analyze network traffic, it can be used to debug any type of data communication by modifying the Passthrough Sector. For instance, PCIe signals can also be converted to AXI Stream and be observed in a similar manner. FFShark can also be used to monitor and debug general signals in an FPGA design.

It is a research question to find if and how the BPF language could be modified to better serve as an interactive debugging tool. For example, eBPF [22] is used in the Linux kernel to track filesystem calls, create histograms of I/O transfers, and other advanced debugging tasks. Not all the features supported by eBPF map well to an FPGA implementation, but the shared memory model would be essential to allow the collection of global statistics at 100G speeds.

The BPF Core is a processor that can accept arbitrary programs[5]. While generic statistics are simple to collect, we wish to create a method for the user to program new types of measurements and have this system report live values. These custom statistics can then be used by SDN applications to improve routing and other decisions.

This work is currently compatible with Wireshark in the sense that it uses the same BPF machine code and returns the same results. However, some work is needed to integrate this with the Wireshark code base so that the familiar Wireshark GUI and utilities may be used. This could be done by creating a custom OS network driver that transfers filter programs to FFShark instead of executing them in the kernel.

## VII. CONCLUSION

We have created a passthrough device, operating at 100G, which can be programmed to flag and store a subset of observed packets. This subset is specified using the PCAP filtering language, allowing users to continue using familiar syntax. The device was also shown to operate at full 100G speeds without dropping packets and was shown to be immediately expandable to 400G once transceiver and FPGA network support is available. The device adds 400ns of latency to the network data, comparable to a commercial 100G switch. Since networked designs expect latencies in this range and FFShark does not significantly change the environment, this justifies its use for debugging in a live environment.

## VIII. ACKNOWLEDGEMENTS

M.A. Merlini and J.C. Vega are equal contributors to this work. The authors would like to thank the anonymous reviewers for their insightful feedback. We thank the Edward S. Rogers Sr. Department of Electrical and Computer Engineering, the Ontario Graduate Scholarship, Huawei, Xilinx, and NSERC for financial support. Finally, we thank Xilinx and Fidus Systems for donating chips, tools, and support.

[5]Some features, such as looping, are explicitly forbidden by the Linux kernel to prevent misbehaved filters from crashing the system. However, we have elected to remove these restrictions

## REFERENCES


[1] Wireshark, "Wireshark User's Guide Version 3.3.0," 2018, https://www.wireshark.org/docs/wsug_html/.

[2] Intel, "Intel Core I9 9900KS processor specifications," 2019, https://www.intel.ca/content/www/ca/en/products/processors/core/i9-processors/i9-9900ks.html.

[3] PCIe-Consortium, "PCI Express® Base Specification Revision 3.0," *PCIe Group*, pp. 1–860, 2010, http://www.lttconn.com/res/lttconn/pdres/201402/20140218105502619.pdf.

[4] M. Merlini and C. Vega, "A versatile Wireshark-compatible packet filter, capable of 100G speeds and higher," 2020, https://github.com/UofT-HPRC/fpga-bpf.

[5] V. Jacobson, C. Leres, and S. McCanne, *pcap-filter(7) - Linux man page*, Lawrence Berkely National Laboratory.

[6] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture." in *USENIX winter*, vol. 46, 1993.

[7] S. Campbell and J. Lee, "Prototyping a 100G monitoring system," in *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. IEEE, 2012, pp. 293–297.

[8] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[9] A. Cardigliano, "PF-RING - nBPF," 9 2019, https://github.com/ntop/PF_RING/tree/dev/userland/nbpf.

[10] D. Luca, "nBPF," *Sharkfest Europe*, pp. 26–37, 2016, https://sharkfesteurope.wireshark.org/assets/presentations16eu/02.pdf.

[11] "Netcope P4," product Brief. [Online]. Available: https://www.xilinx.com/products/intellectual-property/1-pcz517.html

[12] "P4 Language Specification," 2019. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.2.0.pdf

[13] "Full Line Rate Sustained 100Gbit Packet Capture," product Brief. [Online]. Available: https://www.fmad.io/products-100G-packet-capture.html

[14] "10/25/40/100G Packet Broker with PCAP Filtering," product Brief. [Online]. Available: https://www.bittware.com/102540100g-packet-broker-with-pcap-filtering/

[15] Fidus, "Sidewinder-100 Datasheet," 2018, https://fidus.com/wp-content/uploads/2019/01/Sidewinder_Data_Sheet.pdf.

[16] Q. Shen, "100G UDP Link for FPGAs though AXI STREAM (GULF STREAM)," 2019, https://github.com/UofT-HPRC/GULF-Stream.

[17] R. Fung, V. Betz, and W. Chow, "Simultaneous Short-Path and Long-Path Timing Optimization for FPGAs," *IEEE/ACM International Conference on Computer Aided Design*, pp. 838–845, 2004, https://ieeexplore.ieee.org/document/1382691.

[18] J. Vega, Q. Shen, A. Leon-Garcia, and P. Chow, "Introducing ReCPRI: A Field Re-configurable Protocol for Backhaul Communication in a Radio Access Network," *IEEE/IFIP International Symposium on Integrated Network Management*, pp. 329–336, 2019, https://ieeexplore.ieee.org/document/8717902.

[19] J. Vega, "SHIP: A Storage System for Hybrid Interconnected Processors," Master's thesis, University of Toronto, Department of Electrical and Computer Engineering, Apr. 2020.

[20] Mellanox, "100Gb/s QSFP28 Direct Attach Copper Cable," 2018, https://www.mellanox.com/related-docs/prod_cables/PB_MCP1600-Exxx_100Gbps_QSFP28_DAC.pdf.

[21] Xilinx, "UltraScale Devices Integrated 100G Ethernet Subsystem v2.5," 2019, https://www.xilinx.com/support/documentation/ip_documentation/cmac/v2_5/pg165-cmac.pdf.

[22] B. Gregg, *BPF Performance Tools*. Addison-Wesley Professional, 2019.