UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO FIN DE GRADO

ANALYSIS OF XILINX SDNET TOOL FOR PACKET FILTERING IN 100 GBPS NETWORK MONITORING APPLICATIONS

Sergio Fuentes de Uña
Tutor: José Fernando Zazo Rollón
Ponente: Sergio López Buedo

JUNIO 2018

ANALYSIS OF XILINX SDNET TOOL FOR PACKET FILTERING IN 100 GBPS NETWORK MONITORING APPLICATIONS

DISEÑO E IMPLEMENTACIÓN DE FILTROS DE PAQUETES PARA REDES 100 GBPS ETHERNET MEDIANTE LA HERRAMIENTA XILINX SDNET Y LOS LENGUAJES PX Y P4

Autor: Sergio Fuentes de Uña
Tutor: José Fernando Zazo Rollón
Ponente: Sergio López Buedo

Departamento de Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid

JUNIO 2018

Acknowledgment

“Do everything by hand,
even when using the computer.”

– 宮崎駿 (Hayao Miyazaki)

I should like to start by expressing my gratitude to the Ministry of Education, Culture and Sport as well as the UAM Electronic and Communication Technologies Department for supporting this project with a Collaboration Grant (B.O.E. 12th August 2017).

I would like to thank my mentors José and Sergio for introducing me into the FPGA jungle and carefully guiding me through all my work with such kindness, as well as every single person in the HPCN lab who has added their grain of sand to this project: Rafa, Dani, Mario, Gustavo, Tobi, David... Thank you for making this work possible!

Of course I cannot fail to thank my family for their unconditional love and belief in me ever since I was a little boy: my mom Alicia, my aunt Mari Paz, my cousin Ana and my always-favorite cousin Maribel, who left us this year. Rest in peace, Yeye. I love you.

Allow me to add one last line to thank my friends, more like brothers, Raúl and Diego. Thank you for always being there by my side, willing to pick me up every single time I have fallen along all these years, thank you from the bottom of my heart.

Abstract

Abstract

Network traffic monitoring is becoming more and more challenging due to the relentless increase in network speeds. At 100 Gbps, the classical approach of storing all traffic for a later analysis might not be feasible, since the huge volume of data that needs to be saved could make it impossible. Nevertheless, packet filtering allows network monitoring tools to focus on a certain problem, discarding all packets that are not relevant for the analysis and thus easing storage requirements. The high performance and guaranteed line-rate operation of FPGA-based solutions make them optimal for packet filtering at 100 Gbps. However, the effort required by a conventional, HDL-based FPGA development methodology might be prohibitive. To address this problem, in this work we have analyzed the results obtained with the Xilinx SDNet high-level tool for two packet filtering cases. These two filters are related to the monitoring of sites visited by network users and, for both cases, the SDNet designs were able to operate at line rate on actual 100 Gbps Ethernet links. SDNet results were also compared to HDL implementations made by an experienced engineer. Though HDL-based designs allow for reduced latency and resource utilization, SDNet excels in terms of productivity: the description of the most complex filter only takes about 100 lines of SDNet code, that is, significantly less than the HDL counterpart. While pushing the limits of the SDNet architecture, related systems from the field of Queuing Theory were also modeled and studied.

Keywords — sdnet, fpga, packet, filter, processing, 100 gbps, network, traffic, monitoring, high performance, client hello, dns, ascii, data plane, header, payload, queuing, xilinx.

Resumen

Resumen

Monitorizar tráfico de red es cada vez un mayor desafío debido al incesante aumento de las velocidades de red. A 100 Gbps, la estrategia clásica de almacenar todo el tráfico para su posterior análisis puede no ser factible, dado el enorme volumen de datos que se ha de guardar. Sin embargo, el filtrado de paquetes permite que las herramientas de monitorización de red se enfoquen en un problema particular, descartando los paquetes irrelevantes para el análisis y facilitando así los requisitos de almacenamiento. El alto rendimiento y la tasa de línea que brindan las soluciones basadas en FPGA las hacen óptimas para el filtrado de paquetes a 100 Gbps. No obstante, el esfuerzo requerido por la metodología convencional de desarrollo FPGA es en ocasiones problemático. Para abordar este inconveniente, en este trabajo hemos analizado los resultados obtenidos mediante la herramienta de alto nivel Xilinx SDNet para dos casos de filtrado de paquetes. Dichos filtros están relacionados con la monitorización de las páginas visitadas por los usuarios de red y, en ambos casos, los diseños de SDNet fueron capaces de funcionar a tasa de linea en enlaces 100 Gbps Ethernet reales. Los resultados de SDNet se han comparado también con implementaciones HDL realizadas por un ingeniero experimentado. Aunque los diseños HDL logran menor latencia y uso de recursos, SDNet sobresale en términos de productividad: la descripción del filtro más complejo sólo requiere 100 líneas de código SDNet, esto es, significativamente menos que el HDL equivalente. Durante la investigación de la arquitectura de SDNet, también se han modelado y estudiado sistemas de gran interés, pertenecientes al campo de la teoría de colas.

Palabras clave — sdnet, fpga, paquete, filtro, procesar, 100 gbps, red, tráfico, monitorizar, alto rendimiento, client hello, dns, ascii, data plane, header, payload, colas, xilinx.

Contents

List of Tables

List of Figures

Glossary

AXI4-Stream AXI4-Stream is one of many AMBA-based protocols designed to transport data streams of arbitrary width in hardware. Most usually 32-bit bus width is used, which means that 4 bytes get transferred during one cycle. At 100MHz of programmable logic frequency on FPGAs this yields throughput of magnitude of hundreds of megabytes per second depending on memory management unit capabilities and configuration. 9, 10, 16, 19, 30, 31

Data Plane The data plane (sometimes known as the user plane, forwarding plane, carrier plane or bearer plane) is the part of a network that carries user traffic that defines the part of the router architecture that decides what to do with packets arriving on an inbound interface. The data plane, the control plane and the management plane are the three basic components of a telecommunications architecture. The control plane and management plane serve the data plane, which bears the traffic that the network exists to carry. 2, 5–7, 33, 35, 36

FPGA Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing. This feature distinguishes FPGAs from Application Specific Integrated Circuits (ASICs), which are custom manufactured for specific design tasks. 1, 2, 4–8, 17, 20, 21, 33, 35

Acronyms

ASCII	American Standard Code for Information Interchange. 24–26, 30, 31
CDF	Cumulative Distribution Function. 29, 39, 40
COTS	Commercial Off-The-Shelf. 20
CPU	Central Processing Unit. 8, 35
DIY	Do It Yourself. 6
DNS	Domain Name System. 2, 10, 11, 13, 16–18, 31
DPI	Deep Packet Inspection. 11, 23, 30, 31, 33, 35
ECDF	Empirical Cumulative Distribution Function. 29, 39
FIB	Forwarding Information Bases. 8
FIFO	First In, First Out. 12, 19, 26, 30, 31, 34, 35
HDL	Hardware Description Language. 2–4, 7, 9, 10, 12, 17, 19, 20, 31, 33, 35
HLS	High-Level Synthesis. 2
IP	Internet Protocol. 1, 2, 11, 18, 24
IPv4	Internet Protocol version 4. 2
IPv6	Internet Protocol version 6. 2, 11, 16
JTAG	Joint Test Action Group. 10, 13
MLE	Maximum Likelihood Estimation. 28, 29

QoR	Quality of Results. 2, 7, 8, 12
SNI	Server Name Indication. 11, 23
TCP	Transmission Control Protocol. 1, 24
TLS	Transport Layer Security. 2, 10, 11, 13, 16, 17, 20, 23, 31
UDP	User Datagram Protocol. 2, 11, 18, 24
VLAN	Virtual Local Area Network. 2, 11, 16, 18, 24

1 Introduction

It is well known that programmable logic plays a prominent role in the field of high-performance networking. FPGA-based solutions not only provide high processing speeds but also offer small latencies, and what is more important, a deterministic operation. However, in recent years, several software frameworks have been developed to overcome the performance shortcomings of conventional Transmission Control Protocol (TCP) / Internet Protocol (IP) stacks. Among all these frameworks, DPDK is probably the one most widely used [1]. DPDK performs very well in terms of raw networking speeds, being able to reach 100 Gbps on high-end servers. But the problem with software-based solutions is latency and non-determinism. As packets go through a number of hardware and software queues, latency is significantly increased. Moreover, the latency of these queues is non-deterministic, and performance is obtained by splitting incoming traffic into several processing threads, each running in a different processor core. As a consequence, packet disorder might happen [2].

Nevertheless, although FPGA-based solutions for packet processing can obtain better Quality of Results (QoR) than software (DPDK-based) ones, the fact is that software has traditionally beaten FPGA in terms of development costs. To make FPGA competitive in terms of development effort, several high-level synthesis tools have been introduced over the last years, such as Vivado High-Level Synthesis (HLS) and the SDx family [3] by Xilinx, or Intel HLS Compiler [4] by Intel. The goal of all these tools is to increase the abstraction levels in order to make FPGA development much more productive. Not all high-level synthesis tools are general purpose: There are application-specific tools such as the one studied in this work, Xilinx SDNet, which is specifically tailored towards network Data Plane (packet processing) applications.

Network traffic monitoring is certainly one of the fields that can benefit from FPGA-based packet processing. The traditional approach was to capture all traffic for a later analysis. However, this approach is losing validity as the network speeds increase: At 100 Gbps, up to 45 terabytes of traffic can be collected each hour. Fortunately, not all traffic is usually needed, only a small fraction of it is typically relevant for a given analysis. Therefore, packet filtering can be a convenient approach to scale network monitoring to 100+ Gbps speeds. However, this filtering needs to be done very carefully. Firstly, no packet losses are allowed at any case, even in the presence of corner cases such as minimal-size packets. Secondly, it is important that the solution is deterministic, especially in terms of packet order, but it is also important that the latency is bounded, to increase the accuracy of packet timestamps. All these requirements call for a FPGA-based solution.

The purpose of this work is to analyze the QoR obtained by Xilinx SDNet in the development of FPGA-based solutions for packet filtering. As case-study, two different packet filters have been evaluated: Transport Layer Security (TLS) Client Hello and Domain Name System (DNS) Request/Reply. Both are complementary, oriented towards monitoring what are the sites visited by network users. The former needs to inspect the payload of packets, while the latter just inspects the IP and User Datagram Protocol (UDP) headers. But, in order to make the DNS filter more complex, it also supports Virtual Local Area Network (VLAN) tags and both Internet Protocol version 4 (IPv4) and Internet Protocol version 6 (IPv6) protocols.

Results obtained with the SDNet tool are compared with a conventional Hardware Description Language (HDL) implementation made by an experienced FPGA designer. As it will be shown later in the results and conclusion sections, while the resource usage of the SDNet solution is not as good as the one of the HDL designs, the SDNet implementations guarantee 100 Gbps operation, and they feature a moderate latency of around 100 clock cycles. Nevertheless, the most interesting result from SDNet is that description of filters only takes around 120 lines of high-level code. Compared to the approximate 650 lines of Verilog code that filters take for the reference designs, the benefits in terms of productivity are remarkable.

1.1. Scope

The work described in this document is aimed towards hardware and software engineers who are involved in the networking area, specifically those who research development of hardware-accelerated packet filters but find HDL design tedious and counterproductive, or those who simply prefer programming in higher-level languages. The architecture designs, experimental runs, productivity results and successful deployment scenario that follow this introduction will hopefully prove useful for individuals in such field of research when facing similar situations.

The study carried out is also a promising starting point in SDNet development and its underlying PX language. Developers interested in knowing about the capabilities and performance details of this emerging solution, in contrast with similar P4 or traditional HDL approaches to the networking paradigms that SDNet covers, can hence benefit from the thorough analysis of the tool presented in this work.

Simpler routing use cases of SDNet and technical specifications of the PX language remain out of the scope of this document and can be freely consulted in their respective user guides (UG1012 and UG1016) under the SDNet section of Xilinx Design Tools [3].

1.2. Outline

The report is structured in six main sections related to the progress of the project:

Introduction Presentation and motivation of the project carried out and general guidelines about the overall structure of this report document.

Technology Analysis of the state of the art scenario of packet filtering in very high-speed networks, along with the most relevant available solutions and similar published works, considering how they compare with the SDNet approach under study.

Design Exhaustive descriptions of the principal architectures, resources and metrics developed and employed in order to perform the benchmarking of the filters.

Results Experimental results and comparisons between SDNet and HDL obtained from the execution and evaluation of both design alternatives created for the analysis.

Analytical Model Further investigation and experiments about a model of increased complexity, proposed in order to bypass the current limitations of SDNet, which introduces stateful processing and grants access to more ambitious filtering techniques.

Remarks Conclusions and final thoughts on the performed analysis of the SDNet tool and several interesting results that it has brought to light.

Future Work Suggestion of additional studies that might follow the steps taken during this project, such as tool improvements or extensions for even more productive development environments when creating FPGA-based network applications.

2 Technology

2.1. Current Standard Proposals

Nowadays, the high-performance networking sector features a huge number of relevant works, many of which put their effort into creating packet routing and filtering standards, being the latter our main concern when studying SDNet.

The most successful and widely spread solutions are OpenFlow [5] and P4 [6]:

OpenFlow is a communications protocol that grants access to the Data Plane of a network switch or router, enabling advanced packet routing and certain level of filtering. OpenFlow is already supported by many hardware and software routers and switches, representing a great tool for such purposes, but its level of optimization and fine-grain tweaking remains far from what can be achieved by the direct silicon mapping of hardware design in FPGA devices.

P4 conversely, was born as a complete programming language specialized in Data Plane processing. Manufacturers of routing and switching solutions would then build their own compilers that their customers could effortlessly run P4 programs on the corresponding device. The idea behind P4 is very promising and some manufacturers are indeed providing P4 tools for their products, but its increasingly rich grammar makes building P4 compilers a rather tedious task and, in fact, they often only support a small subset of the whole language –notice that not all software coding structures are hardware-synthesizable (e.g. time-awareness)–.

Within this fashion, Xilinx decided to create their own language called PX [7] instead of simply providing tools for P4 development. This decision involves technological dependence, but greatly favors the integration and smooth design flow between the different tools and boards built by Xilinx, offering FPGA-based network applications with very reduced time-to-market when using Xilinx technologies.

The following table presents the technical differences found between the PX and P4 languages, allowing for a fair more detailed comparison:

Table 2.1: Technical Comparison between the PX and P4 Languages

PX	P4
Bundled compiler implementation	Do It Yourself (DIY) compiler implementation
Simplified grammar	Rich grammar (with subsets)
C++ inspired syntax	Java inspired syntax
Does not support libraries	Supports libraries
Designed for 100+ Gbps networks	Contingent performance

It is worth mentioning that SDNet actually includes a supplementary P4 compiler, which translates programs written in a P4 subset into their equivalent PX descriptions that are then ready to be compiled by SDNet into synthesizable Verilog code. UG1252 in the SDNet section of Xilinx Design Tools [3] refers to this P4-SDNet Translator.

2.2. Related Work

This section focuses on documenting similar existing proposals and how they particularly compare with the SDNet analysis introduced during this project.

2.2.1. Automated tool for generating packet filters

In a previous work done by the HPCN Lab [8], a tool that procedurally generates HDL code for packet filters is presented. The tool is tailored towards the 100 Gbps Ethernet interfaces of Xilinx UltraScale/UltraScale+ devices, and defines a simple grammar for specifying filters. Starting from a description that follows said grammar, it generates highly-optimized synthesizable SystemVerilog code. Actually, the very narrow scope of the tool (packet filters) allows for a significantly better QoR than the one that could be obtained from a general-purpose tool.

2.2.2. P4-to-VHDL

P4-to-VHDL [9] follows a similar methodology to the one used in this project: Using a high-level language to describe a Data Plane application, with the aim of generating HDL code that can be implemented in an FPGA device (though they respectively use P4 and VHDL). This project actually performs an intermediate step, in which P4 code is reinterpreted for HFE-M2 [10], a low-latency modular packet header parser architecture for FPGA. The final output of this tool is a P4-equivalent VHDL design than can later be mapped to hardware. The workflow is able to reach full line-rate operation in 100 Gbps Ethernet networks [11].

2.2.3. P4FPGA

Another akin approach is P4FPGA [12], in which a conventional P4 compiler produces mid-level code written in Bluespec BSV language [13]. This BSV specification is afterwards ready to be converted into synthesizable Verilog code by means of the BSC compiler.

2.2.4. Whippersnapper

Regarding tools analysis, Whippersnapper [14] is a recent proposal that describes a benchmarking suite for P4 compilers, including P4FPGA and P4-SDNet. Whippersnapper focuses mainly on implementation and runtime details of the compilers, while our work aims at productivity and QoR of the hardware implementations.

2.2.5. HyPaFilter

HyPaFilter [15] is an hybrid hardware-software system that includes a NetFPGA board and a Linux host. In this work efforts are aimed towards creating a high performance firewall, by taking advantage of the fast routing capabilities of hardware designs. The system handles different user-defined packet processing policies, and filters network traffic using the hardware that has been implemented in the FPGA, though in the most complex cases it has to be assisted by a general-purpose Central Processing Unit (CPU) running Linux. While such scenario is inherently different from the one discussed in this document, it highlights the positive impact of FPGA implementations for network filtering applications, firewalls in this case.

2.2.6. "Matching circuits can be small"

Finally, [16] shows how Forwarding Information Bases (FIB) held by routers can be highly optimized in hardware and therefore easily implemented in FPGA-based device due to the nature of their circuitry. This work gives rise to the fact that hardware-accelerated network packet routing is a matter of concern with several solutions already available in the market and even running in production. But once again, filtering requires more complicated architectures than routing and are not solved that easily, which brings to light the main reason behind this work.

2.2.7. NetFPGA SUME

On a more general FPGA-related line of work aimed towards networking, a community hardware project [17] offers an FPGA-based PCI Express board with I/O capabilities for 100 Gbps operation as a network interface card, multiport switch, firewall, or test and measurement environment. These are some of the capabilities that SDNet aims for, however, the final product does not feature an actual 100 Gbps interface, as opposed to modern Xilinx boards like the one we will be using in this work. Therefore, this open hardware approach remains one step behind the bleeding-edge network interfaces that have already been available in the market for some time now.

3
Design

3.1. Hardware Design

In order to evaluate results in an scenario as close as possible to an eventual production environment, performance measurements have been performed in real hardware. Such measurements required the development of a number of components, which are following described. The block diagram of the design used in the experiments is shown in figure 3.1.

SynthGen HDL implementation of a synthetic traffic generator that outputs a sustained stream of similar packets of chosen size to an AXI4-Stream interface. Supports runtime byte-level packet customization and can be eventually substituted by an actual 100 Gbps Ethernet interface.

BW Meter HDL module that measures bandwidth by keeping track of the number of bytes and packets that went through an AXI4-Stream interface and the number of clock cycles that have elapsed.

LAT Meter HDL module that measures latency by counting how many clock cycles elapse from the moment that a parameterizable amount of packets enter an external module until they exit it.

Filter Module under test that contains the implementation of the packet filter and uses AXI4-Stream interfaces.

Null Sink Terminal AXI4-Stream receiver that holds the TREADY signal continuously high and discards all incoming data.

Metrics Monitor External element that gathers the statistics reported by the BW and LAT Meters, allowing real-time reads through the Joint Test Action Group (JTAG) port of the device (using Vivado Integrated Logic Analyzer).

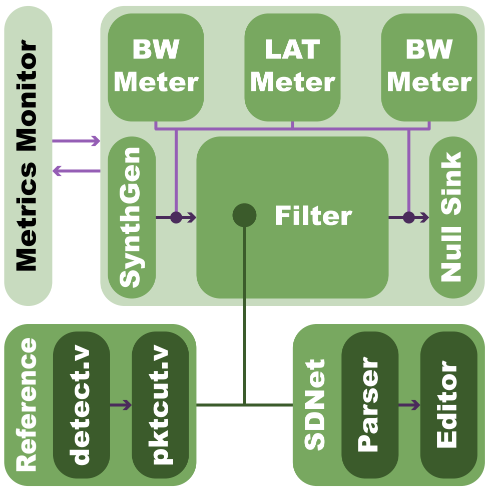

Figure 3.1: Hardware Block Diagram.

As it can be seen in figure 3.1, both filter implementations (Reference –corresponding to the conventional HDL approach– and SDNet) split functionality into two blocks that are later explained: A first one for packet inspection and evaluation of the conditions, followed by a second one for discarding the packets if necessary.

Clock frequency was 322.265625 MHz, and the width of AXI4-Stream buses was 512-bit. These parameters guarantee 100 Gbps line rate operation.

3.2. Implemented Filters

As it was stated in the introduction, two case-study filters were considered: TLS Client Hello and DNS Request/Reply.

- The TLS Client Hello filter looks for packets containing the TLS “Client Hello” message. This kind of packets is interesting because the Server Name Indication (SNI) extension of the “Client Hello” message provides the name of the server to which the client wants to establish a TLS session. The filter requires payload inspection to look for TLS packets, and particularly, those containing the “Client Hello” message
- The DNS Request/Reply filter identifies DNS requests and replies. UDP Packets whose source or destination port is 53 are considered to be DNS messages. That is, the filter only requires IP and UDP header inspection. In order to add some complexity to the filter, it also supports up to two nested VLAN tags and IPv6.

These two filters provide information about the name of servers to which clients are connecting. DNS information would in principle be enough, but it might happen that DNS responses are cached in clients (especially if the value of the TTL field of DNS Resource Records is high). In that case, there will be no DNS request, but the contents of the SNI extension of the TLS “Client Hello” message can be very useful to find out which is the site requested by the client.

An additional Deep Packet Inspection (DPI) filter, which is to be separately detailed later in chapter 5 due to its singular complexity, was also implemented using SDNet.

Two variants of each filter were implemented: One obtained through synthesis of the reference Verilog code, and the other using SDNet 2017.3 to compile the description of the filter written in PX language. Table 3.1 details the number of effective lines of code required to describe the filters in each of the mentioned languages [1].

Table 3.1: Lines of Code Required for Each Implementation

Filter	Verilog Lines of Code	PX Lines of Code
Client Hello	633	118
DNS	645	117
DPI	970	256

[1]With the intention of favoring the reproducibility of the results, the complete code of both Verilog and SDNet (Appendix C) filter implementations is available at `https://github.com/Serede/sdnet-filters`

3.2.1. Reference HDL Filter Architecture

In order to evaluate the QoR of SDNet designs, equivalent filters purely written in Verilog HDL have been developed. These Verilog designs are composed by two main modules: `detect.v` and `pktcut.v`. The former inspects both protocol headers and payload of packets, eavesdropping for particular patterns. The latter drops the whole packet frame when the hardcoded conditions previously checked by `detect.v` are not satisfied. This is depicted in Figure 3.2.

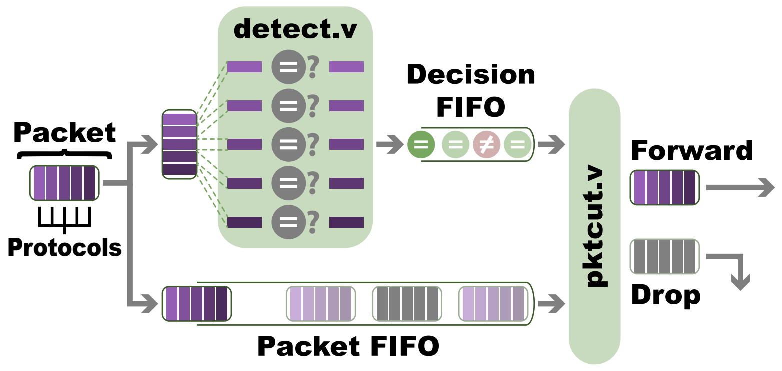

Figure 3.2: Reference HDL Filter Scheme.

The implementation of the module `detect.v` is conceptually straightforward, though it is unavoidably cumbersome to adapt it to new protocols and configurations. For every packet, all filtering conditions are tested in parallel, to achieve both high performance and low latency. A comparison between this approach and the one followed by SDNet one is discussed in the subsequent section.

Once the contents of the relevant fields of the packet have been checked against the filtering rules, `detect.v` generates a single-bit decision flag that indicates whether the packet must be forwarded or dropped. At the same time, the packet is stored in a First In, First Out (FIFO) structure, waiting for the outcome from `detect.v` to be available. Once that `detect.v` has finished, `pktcut.v` forwards or drops the packet according to the aforesaid decision flag.

The latency of the implemented solution is minimal, and most of the clock cycles are spent in storing the packet before it is forwarded to the next component. Only 2 cycles are required by the `detect.v` module, since it performs all checks in parallel.

3.2.2. SDNet Filter Architecture

SDNet designs are essentially a collection of different specialized engines connected in cascade. Each engine solves a certain problem (parsing, editing, etc.) and can be extensively configured to fit the requirements of the network processing application that is being implemented. In this way, packets and tuples (metadata) flow through the datapath as they are being processed by the subsequent engines. This behavior allows designers to easily create network filters.

In the scenario considered in this work, the course of action is to parse each packet against a certain condition (TLS Client Hello and DNS) and discard those packets that do not meet such condition. Translating that into SDNet architecture, an initial *Parser Engine* tests the filter condition and delivers the result to an *Editor Engine*, which accordingly forwards or discards the packet.

As a final remark, these two implementations (Verilog and SDNet) are entirely equivalent and have similar port interfaces. Therefore, they can be perfectly swapped with each other in the final design.

3.3. Sample Packets

The synthetic traffic generator from Figure 3.1 uses TLS Client Hello and DNS Request/Reply packets obtained from a real network trace. This generator has the capability of altering certain key bytes of the packet at runtime, to enforce that packets are eventually filtered out or not. As it will explained in section 4.7, the synthetic traffic generator is used to measure the throughput and latency of filters, but in the production designs it will be substituted by the actual 100 Gbps Ethernet network interface.

3.4. Target Board and Experiments

Designs were synthesized and implemented using the Xilinx Vivado 2017.4 Design Suite, targeting the Xilinx Virtex UltraScale VCU108 Evaluation Kit [18] seen in figure 3.3. Once the whole setup was verified and completely functional, measurements were ready to be gathered from the design, using Vivado Logic Analyzer and the JTAG port of the device. Live tests with synthetic traffic at a sustained 100+ Gbps data rate were performed. The duration of each test was 1 minute; several tests were launched, varying the percentage of filtered-out packets in each test from 10% to 90%.

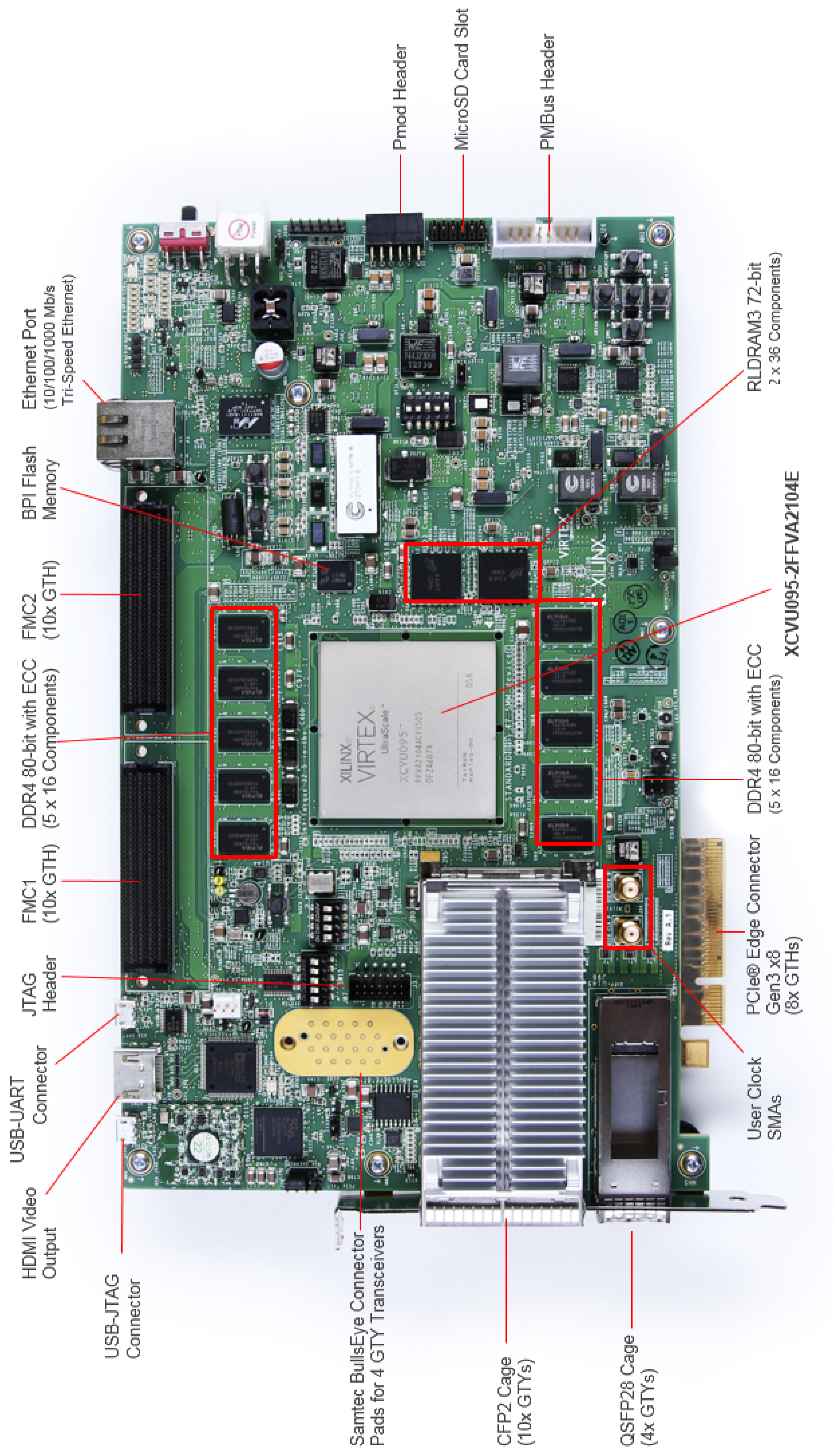
Pmod Header
MicroSD Card Slot
PMBus Header
Ethernet Port (10/100/1000 Mb/s Tri-Speed Ethernet)
BPI Flash Memory
FMC2 (10x GTH)
DDR4 80-bit with ECC (5 x 16 Components)
FMC1 (10x GTH)
JTAG Header
USB-UART Connector
HDMI Video Output
USB-JTAG Connector
Samtec BullsEye Connector Pads for 4 GTY Transceivers
CFP2 Cage (10x GTYs)
QSFP28 Cage (4x GTYs)
User Clock SMAs
PCIe® Edge Connector Gen3 x8 (8x GTHs)
DDR4 80-bit with ECC (5 x 16 Components)
XCVU095-2FFVA2104E
RLDRAM3 72-bit 2 x 36 Components)

Figure 3.3: Xilinx Virtex UltraScale FPGA VCU108 Evaluation Kit.

4 Results

Results obtained from the experiments show that SDNet designs are capable of operating at 100 Gbps, revealing at the same time some drawbacks when compared to the reference designs written in Verilog.

4.1. Bitrate

The first metric analyzed was the maximum bitrate attainable for each design, for the worst case (0% packets filtered). It is worth noting that both the reference and the SDNet designs achieved the same value (see Table 4.1).

Table 4.1: Filter Bandwidth and Latency Comparison

Filter	Bandwidth in Gbps		Latency in Cycles	
	Reference	*SDNet*	*Reference*	*SDNet*
Client Hello	150.305	150.305	34	93
DNS	99.258	99.258	18	101

Actually, a more detailed analysis indicates that these values correspond to filters always accepting new data each clock cycle (that is, the AXI4-Stream TREADY signal of their input interfaces never goes low): Firstly, the raw bandwidth of the 322.265625 MHz, 512-bit wide AXI4-Stream interfaces is 165 Gbps. Secondly, packets are always aligned with the start of an AXI4-Stream 512-bit (64-byte) word. That is, if a packet does not completely fill the last word, this word cannot be used for the following packet. Hence, for a 583-byte long TLS Client Hello packet, ten 64-byte AXI4-Stream words will be needed (that is, 640 bytes). Similarly, for a 77-byte long DNS packet, two 64-byte AXI4-Stream words will be needed (i.e. 128 bytes). As a result, the maximum theoretical bandwidth for each kind of packet is:

$$165 \text{ Gbps } \times 583/640 \text{ Effective Bytes} = 150.3046875 \text{ Gbps}$$
$$165 \text{ Gbps } \times 77/128 \text{ Effective Bytes} = 99.2578125 \text{ Gbps}$$

Which matches the empirical results presented in Table 4.1. The conclusion is that bitrate is limited by the AXI4-Stream bus and the packet alignment requirement, not by the filters themselves. Finally, it is worth noting that these bandwidth values are higher than those present in real 100 Gbps Ethernet links. The benefit of testing with a synthetic traffic generator is being able to stress circuits above real operation conditions.

4.2. Latency

Latency results in Table 4.1, despite being satisfactory for both the reference and SDNet designs, demonstrate the inherent architectural differences between both solutions. We have observed that latency remains constant over the varying percentage of filtered-out packets, a behavior caused by the strictly pipelined architectures used by both solutions. However, SDNet leads to increased latencies –93 and 101 versus 39 and 24, respectively–. We believe that this circumstance is mainly due to the compilable nature of the PX language. The reason why latency is specially increased in the DNS filter, even though the filtering condition itself is simpler, is that additional nested VLAN and IPv6 protocols force the SDNet compiler to add extra layers to the final architecture.

4.3. Hardware Utilization

Figure 4.1 shows a detailed view of the resource utilization of the two alternative solutions, reference (purple) and SDNet (green). The number of FPGA resources displayed is the mean between the two implemented filters (TLS Client Hello and DNS). As differences in resource occupation between the two filters were small, we preferred to show the mean in order to increase the readability of the figure. The main conclusion here is that the number of resources used by the SDNet implementations is typically one order of magnitude bigger than the one for the reference implementation.

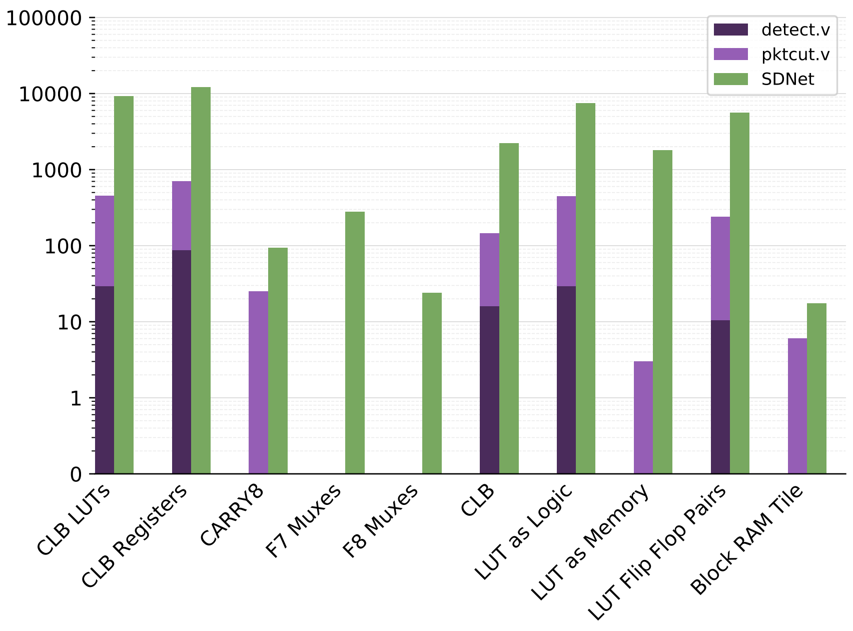

Figure 4.1: Hardware Utilization Comparison (Mean for Both Filters).

4.4. Productivity

As opposed to resource utilization, Table 3.1 shows that SDNet descriptions of filters need 5 times less lines of codes than their Verilog counterparts. Therefore, it can be said that SDNet significantly increases productivity when compared to a traditional HDL-based methodology. Additonally, SDNet automatically creates software C++ testbenches for its modules, making debugging easier.

4.5. Abstraction

Moreover, the much higher level of abstraction that SDNet provides leads to code-bases with improved readability and easier to understand at first sight. For example, take the following code fragment from the reference Verilog implementation of the DNS filter:

Listing 4.1: Verilog Code Excerpt for DNS Filter

```
assign ipv6_port53_nested_vlan =
data[OFFSET_UDP_IPV6+2*VLAN_LEN+:16] == 16'h3500 ||
data[OFFSET_UDP_IPV6+2*VLAN_LEN+16+:16] == 16'h3500;
...
rule <= (ipv4 & udp_ipv4 & ipv4_port53)
| (vlan & ipv4_vlan & udp_ipv4_vlan & ipv4_port53_vlan)
| (vlan & nested_vlan & ipv4_nested_vlan & udp_ipv4_nested_vlan &
    ipv4_port53_nested_vlan)
| (ipv6 & udp_ipv6 & ipv6_port53)
| (vlan & ipv6_vlan & udp_ipv6_vlan & ipv6_port53_vlan)
| (vlan & nested_vlan & ipv6_nested_vlan & udp_ipv6_nested_vlan &
    ipv6_port53_nested_vlan);
```

The first assignment stores whether the destination or source port of the UDP packet is 53. Together with preceding similar assignments in which fields of VLAN, IP and UDP headers are checked, a final reduction of all these conditions is computed and sent through the output rule, which reflects the filter result. Particular attention must be paid about the endianness of literals (`16'h3500` for port 53). This code is perfectly functional and performance-wise optimal, but it is undeniably far from being easily readable and comprehensible.

Now take the analogous code excerpt from the SDNet implementation of the same DNS filter, in this case written in the high-level PX language:

Listing 4.2: SDNet Code Excerpt for DNS Filter

```
// DNS_Parser
class ETH :: Section(1) {
  // ETH can be followed by VLAN, IPV4 or IPV6
  map types {
    (VLAN_TYPE, VLAN), // const VLAN_TYPE = 0x8100
    (IPV4_TYPE, IPV4), // const IPV4_TYPE = 0x0800
    (IPV6_TYPE, IPV6), // const IPV6_TYPE = 0x86dd
    done(SUCCESS)
  }
  ...
} // ETH Header
class VLAN :: Section(2:3) { ... } // VLAN Header
class IPV4 :: Section(2:4) { ... } // IPV4 Header
class IPV6 :: Section(2:4) { ... } // IPV6 Header
class UDP :: Section(3:5) {
  ...
  method update = {
    tuple_out.is_dns = (srcport == 53) ||
                       (dstport == 53)
  }
} // UDP header
```

Another code example is that of the packet discarder. The following piece of code belongs to the *Editor Engine* that has been included in the PX implementations of both filters as an SDNet system builtin:

Listing 4.3: SDNet Code Excerpt for Packet Discarder

```
class FETCH :: Section(1) {
  // Drop only non-DNS packets
  method move_to_section =
    if (tuple_in.is_dns == 0) DROP
    else done(SUCCESS);
} // FETCH
class DROP :: Section(2) {
  // Remove whole packet
  method remove = drop();
} // DROP
```

The equivalent HDL code needed to discard packets from an AXI4-Stream interface (`pktcut.v` from Figures 3.1 and 3.2) requires manually instantiating different FIFO structures for packets and decision flags and writing several HDL processes in order to perform the very same action of dropping certain packets.

4.6. Limitations

While developing filters in PX, intrinsic limitations were brought to light when trying to process complex payloads. The way state machines are internally instantiated in SDNet renders it impossible to travel more than 64 different states throughout the machine. Hence, when recurrently parsing protocols based on the thorough repetition of small structures (like TLS records), the standard engines are tied to this limitation and coding an HDL User Engine becomes necessary in order to circumvent the issue.

4.7. Interoperability Testbed

Up to this point, we have focused on the feasibility of FPGA network filters and how Xilinx SDNet may help hardware engineers towards tackling the problem. However, during the development of state-of-the-art network accelerators, one of the main concerns is the portability of the solution. Not only the standalone capabilities of the filters have been evaluated but also its integration with third party network interface cards.

Both reference and SDNet designs of the filters were additionally ran in a physical testbed featuring a Xilinx VCU108 board and a Commercial Off-The-Shelf (COTS) server. The server equips two Intel Xeon E5-2630 processors running at 2.20GHz and a total of 128GB of main memory. The 100 Gbps endpoint is provided by a Mellanox ConnectX-5 card from the MT27800 family. Figure 4.2 displays the complete testbed. The host operating system used for the experiments was a CentOS 7 Linux distribution thoroughly configured with the set of libraries supplied by DPDK 17.05 [1].

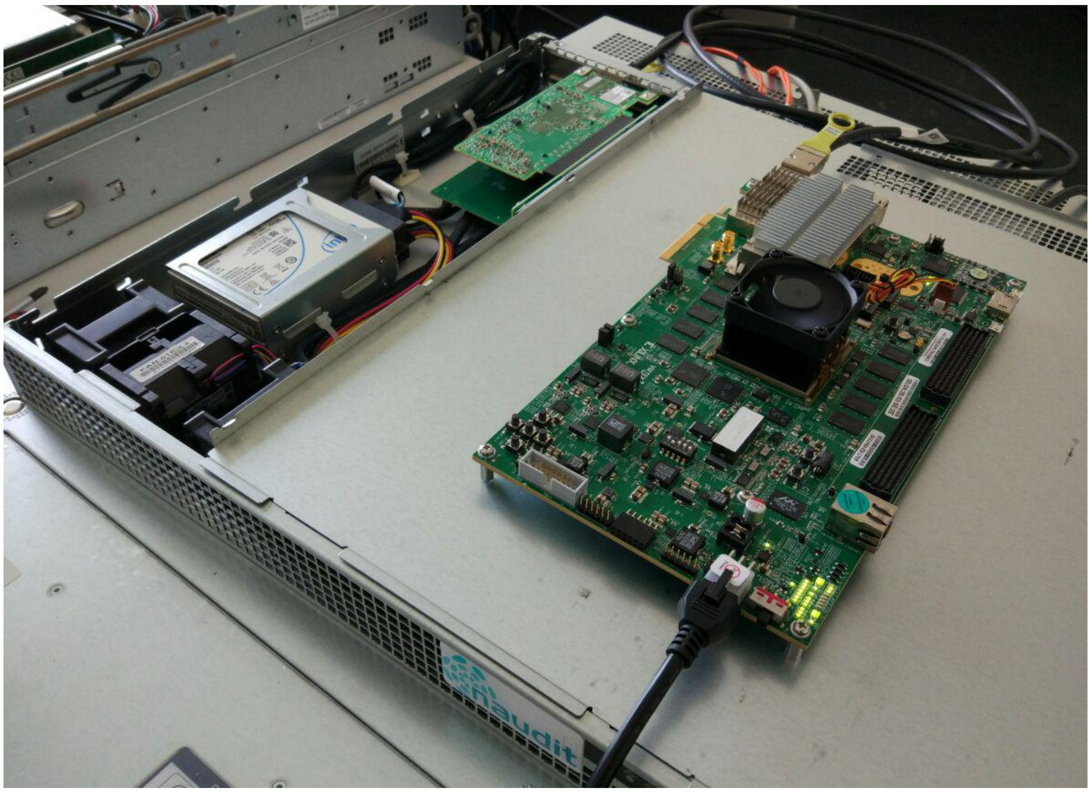

Figure 4.2: Interoperability Hardware Testbed Equipment.

Reception and transmission of packets were verified:

- First, the setup was tested using the synthetic generator in the FPGA platform, able to saturate the 100 Gbps interface. Packets were received flawlessly at the server during the experiments with sustained full line rate.
- Then, the synthetic generator was replaced by the UltraScale Integrated 100G Ethernet Subsystem. Since at the time of writing, open source solutions able to saturate 100 Gbps links with the aforementioned server configuration do not exist, network traffic replay speeds were in this case limited by DPDK (to approximately 70 Gbps). The filters kept operating at the served line rate without issue.

5 Analytical Model

5.1. Motivation

Given the limitations of SDNet aired during chapter 4, the design of more complex packet filters becomes non-trivial when approached from the PX language paradigm.

One of the most interesting scenarios in network traffic filtering is DPI, which requires the packet processor to traverse the whole stack of packet headers down to the payload, where a certain condition is checked. Such process has proven to be noticeably intensive for the processing unit and consequently poses a challenge to the implementations of state of the art deep packet processors.

More specifically, classifying packets between encrypted data and plain data can be of enormous utility when handling very high-speed network traffic. Internet users are becoming more and more concerned about security and the mean percentage of encrypted data in all kinds of networks is steadily increasing. As a consequence, while encrypted traffic analysis is very limited to techniques similar to the previously mentioned TLS Client Hello SNI field, plain text traffic can be thoroughly examined using DPI to look for certain keywords, patterns or even regular expressions.

5.2. Filter Design

For such purpose, based on a previous work from this laboratory [19], a simple yet powerful estimation of whether a packet is encrypted or not can be achieved by checking if each byte of the payload is within the range of printable American Standard Code for Information Interchange (ASCII) characters (i.e. decimal values from 32 to 126), looking for either bursts of consecutive ASCII characters or an overall proportion of ASCII characters in the payload larger than defined thresholds.

In order to implement the aforementioned filter in SDNet, an initial approach was implemented using three cascaded engines as displayed in figure 5.1.

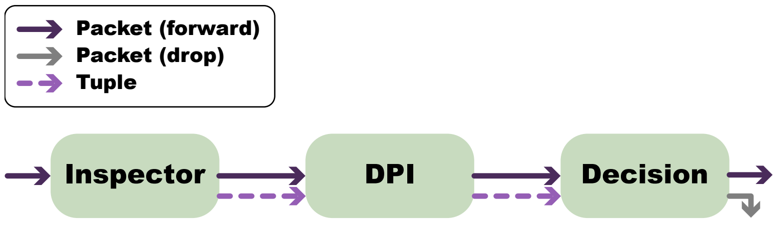

Figure 5.1: DPI Filter Diagram.

Inspector A first Parser Engine in charge of locating the payload in the packet and determining its precise length by going through the different headers (Ethernet, VLAN, IP and TCP/UDP), codifying this information into an output tuple.

DPI A second Parser Engine responsible of jumping to the payload using an input tuple with the information gathered by the previous engine and then performing the bytewise logic described above in order to find ASCII burst and proportion values. These values are also stored in a tuple and forwarded to the last engine.

Decision A final Editor Engine that ultimately drops all packets with ASCII burst and proportion values below the specified thresholds. Notice that this engine could be triggered as soon as either value exceeds its threshold and this may happen at an arbitrary point along the total payload length.

At first glance, this model represents a possible architecture that would perform the desired filter using the programming constructs provided by the PX language and therefore should be able to be compiled into Verilog by the SDNet compiler. However, as it was already mentioned in the Limitations section of the preceding chapter, SDNet codifies its programs as limited state machines that only support transitioning between up to 64 states during their execution flow.

Due to this limitation, the bitwise verification of the printable ASCII range performed by the *DPI* Parser Engine is only executed for the first 63 bytes of the payload (1 state is used for payload positioning) and then the SDNet module simply terminates and throws an error through the control port.

To circumvent such behavior, several modifications were included in the model:

- The three different pieces of logic corresponding to the *Inspector*, *DPI* and *Decision* engines were split and individually implemented.
- The *DPI* module now internally handles graceful termination when it reaches the limit of 64 states and specifically parses no more than 63 bytes of each packet it processes.
- Tuple interfaces were homogenized between the different modules and now include fields for retaining payload information from the *Inspector* engine and keeping track of local counters when computing ASCII burst and proportion values.
- Recirculation was enabled for packets and their respective tuples in the *DPI* engine. Every time a packet enters the *DPI* module, it updates the ASCII burst and proportion values from the next 63 bytes at the current packet offset and increases such offset with the number of bytes processed.
- The *Decision* engine recieves packets with their respective tuples from the *DPI* engine and makes the pertinent decision from the current values in the tuple:
    - If the ASCII burst or proportion values are above their respective thresholds, the packet is marked as plain traffic and forwarded out of the system.
    - If the values are below the thresholds and the end of the packet was reached, the packet is marked as encrypted traffic and dropped from the system.
    - In any other case, the packet is marked for recirculation so that the next 63 bytes are processed when it is fed to the *DPI* engine again.

These tweaks in the architecture allow us to bypass the internal limitations of SDNet by extracting the critical part of the intra-module routing logic to a higher level of the design, controlled now in the Vivado Design Flow. The new model is shown in figure 5.2.

5.3. Stochastic Queuing Model

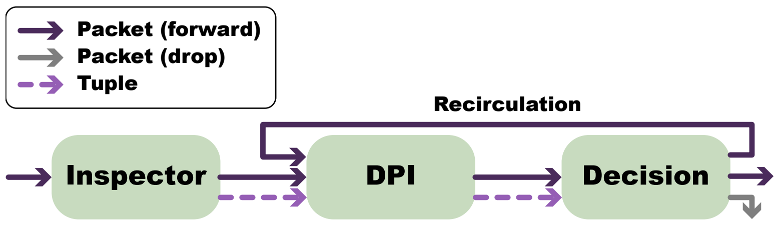

Figure 5.2: DPI Filter with Recirculation Diagram.

This new architecture of the system, however, introduces a key factor into the model: the former deterministic latencies found as a consequence of the linear cascaded structure of SDNet are no longer valid. The recirculation logic is now an stochastic process since it depends on the characteristics of each packet that influence the number of passes through the *DPI* engine it needs (payload size, location of the ASCII burst).

Due to this lack of determinisim, the necessity of an input FIFO before the *DPI* module to store recirculating packets and tuples becomes a cause of concern since it must be correctly dimensioned so that the optimization level of the final system does not get degraded while it remains able to guarantee line rate operation without saturating and discarding packets.

Fortunately, Queuing Theory exists as the mathematical field specialized in modeling this particular type of stochastic processes that involve queuing the elements that flow within the system (in this case, packets and their corresponding tuples of metadata).

Moreover, the cornerstone of Queuing Theory, Little's Law (John Little, 1961), states the proved relationship between the number of elements L in a stationary system and the effective values of arrival λ and service μ rates of such system. A modern formal proof of Little's Law can be found later in appendix A.

The main objective of this analytical model, which is finding the correct dimensioning value for the input FIFO, can therefore be easily estimated from the L value outputted by Little's Law in the proposed system. Nevertheless, directly using L as the FIFO length would cause packets losses in peak periods where the queue becomes full, but dividing Little's result by an overdimensioning tolerance factor α is enough to ensure average reduced FIFO occupancy (e.g. FIFO length $= L/\alpha$, with $\alpha = 60\%$ of average occupancy).

5.3.1. Model identification

Before choosing the model, we need to specify the different stochastic variables present in the queuing system and the notation for their mean values. Following the most common convention employed in Queuing Theory, these variables are:

Table 5.1: Variables and Mean Values for the Queuing Model

$A :=$ time between arrivals. $\mathbb{E}(A) = T_a = \frac{1}{\lambda}$
$S :=$ service time. $\mathbb{E}(S) = T_s = \frac{1}{\mu}$
$T_q :=$ time spent in the queue $\mathbb{E}(T_q) = W_q$
$T :=$ time spent in the system $\mathbb{E}(T) = W = W_q + T_s$
$N_q :=$ number of items in the queue $\mathbb{E}(N_q) = L_q$
$N :=$ number of items in the system $\mathbb{E}(N) = L$

Furthermore, Kendall's notation is the standard system used to describe and classify a queueing node. The short three-factor form of Kendall's notation that we will be using was proposed by D. G. Kendall in 1953 and utilizes the following construction:

$$\underbrace{A}_{\substack{\text{Stochastic process}\\ \text{followed by}\\ \text{item arrival.}}} \quad / \quad \underbrace{S}_{\substack{\text{Stochastic process}\\ \text{followed by}\\ \text{service time.}}} \quad / \quad \underbrace{c}_{\substack{\text{Number of}\\ \text{service channels}\\ \text{in the model.}}}$$

First approach: Feedback model

Since the proposed model includes output feedback (recirculation of packets and tuples), a first possible approach would be to find the overall mean possibility of recirculating each packet and including this feedback in the queuing model as shown in figure 5.2. Additionally, the most interesting and reliable tool to study queuing models with feedback, Jackson's Theorem, requires the arrival process to be a Poisson process as one of its hypothesis. The corresponding Kendall's notation in this ideal model would be:

$$M/D/1$$

Where:

M represents a Poisson arrival process (i.e. exponential inter-arrival times).

D represents a degenerate distribution of the service time. This is an aftermath of the deterministic latency of a single pass through the pipelined SDNet filter.

1 represents the single service channel present in our system: the *DPI* module.

In order to evaluate the validity of this first model, we measured the arrival process under different common scenarios of network traffic, like intranets of big technology companies or the teaching laboratories of the university.

Such task involves gathering timestamp difference samples from a very large amount of packets and feeding them to an statistical tool which corroborates whether they follow a Poisson process or not. In this case, we opted for a Kolmogorov-Smirnov test (appendix B) against the hypothesized exponential distribution of arrivals. More formally:

$A_1, \dots, A_n :=$ independent samples of the stochastic variable A (time between arrivals).

Null Hypothesis $H_0 : A \sim Exp(\hat{\lambda})$

Notice that here $\hat{\lambda} = 1/\bar{A}$ is the Maximum Likelihood Estimation (MLE) for the λ parameter of the exponential distribution (i.e. the inverse of the sample mean).

The following results conform the output for the Kolmogorov-Smirnov test executed in one of the largest traffic scenarios described above:

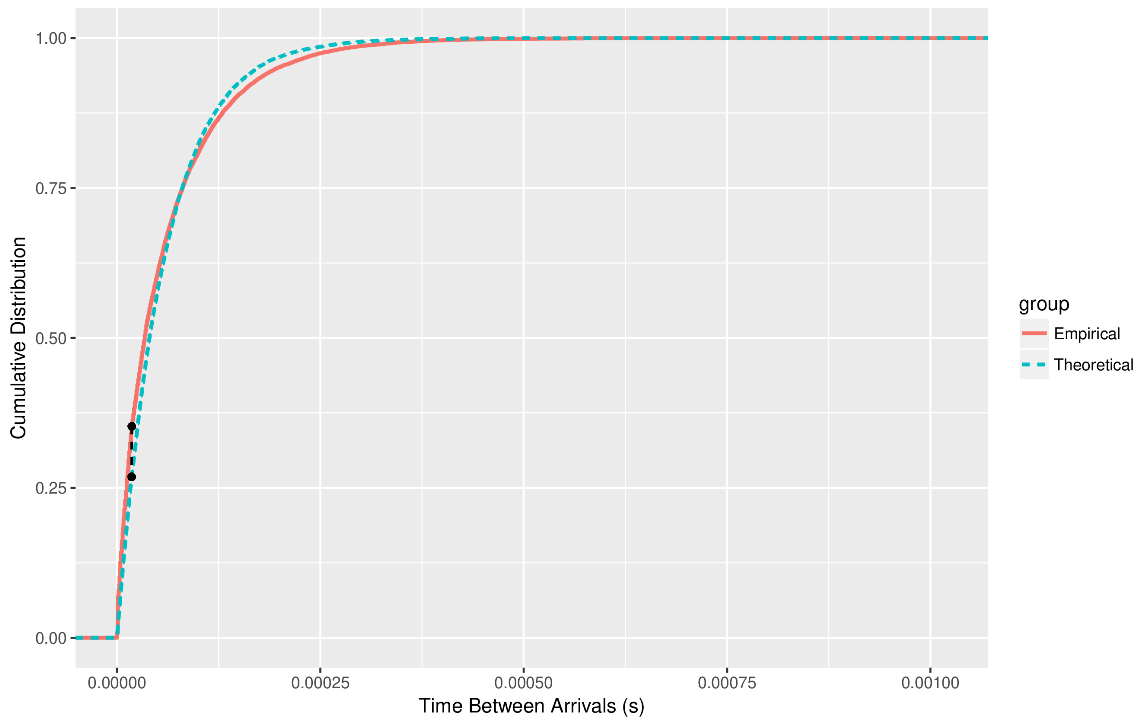

Figure 5.3: Kolmogorov-Smirnov Test for the Distribution of Time Between Arrivals.

From figure 5.3 we can see how, even though the Empirical Cumulative Distribution Function (ECDF) of the variable A is sitting rather close to the Theoretical Cumulative Distribution Function (CDF) of an exponential stochastic variable with the corresponding $\hat{\lambda}$ value obtained from MLE, the maximum difference between both functions found by the Kolmogorov-Smirnov method (the black dashed line) reaches a significant magnitude –roughly 0.1–.

To be more precise, this particular Kolmogorov-Smirnov test returned a p-value of $2.2 \cdot 10^{-16}$, meaning that our null hypothesis $H_0 : A \sim Exp(\hat{\lambda})$ can be rejected with an accuracy level over 99.99%. Thus, there is statistical evidence against the time between arrivals A following an exponential distribution in our cases of study.

This outcome is in fact not surprising since it is well known that arrival times between network packets in a production environment is a long way from being a Poisson process [20]. Actually, the literature on this particular question is enormously rich and diverse, to the point of using fine-grain tweaked Lèvy alpha-stable distributions to model the aforementioned phenomenon in some of the most recent researchs.

Alternative approach: General model

As a consequence of the previously discussed issues of the $M/D/1$ model, a more reasonable suggestion for the queuing model developed in Kendall's notation is:

$$G/G/1$$

Where:

G represents a general distribution of inter-arrival times. This enables shaping the incoming traffic simply from its mean and standard deviation values, without incurring in mistakes caused by assuming a particular known distribution.

G represents a general distribution of service time. Similarly, using this setting includes the feedback effect in the model using its mean and standard deviation values which take into account the mean probability and number of recirculations, instead of delegating this task to Jackson's Theorem.

1 still represents the single service channel present in our system: the *DPI* module.

5.3.2. FIFO length estimation

Back to the original problem, the main concern here is to find L_q in order to determine the mean number of items in the queue and dimension the input FIFO accordingly. Using the queue version of the already mentioned Little's Law makes it possible:

$$L_q = \lambda W_q = \lambda(W - T_s)$$

Now, introducing α as the guaranteed mean FIFO occupancy rate as it was suggested earlier, we obtain a closed formula for the new variable $\mathcal{F}$ (FIFO length):

$$\mathcal{F} := \text{FIFO length} = \frac{L_q}{\alpha} = \frac{\lambda W_q}{\alpha} = \frac{\lambda(W - T_s)}{\alpha}$$

This expression provides dimensioning values for the input FIFO of the designed DPI system from the empirical values of λ, W and T_s –given by the $G/G/1$ model–, which are contingent upon the network traffic being processed due to the presence of recirculation.

5.4. Hardware Implementation

The final hardware-specific design proposed for the developed DPI filter and all its associated digression follows the component scheme displayed in figure 5.4.

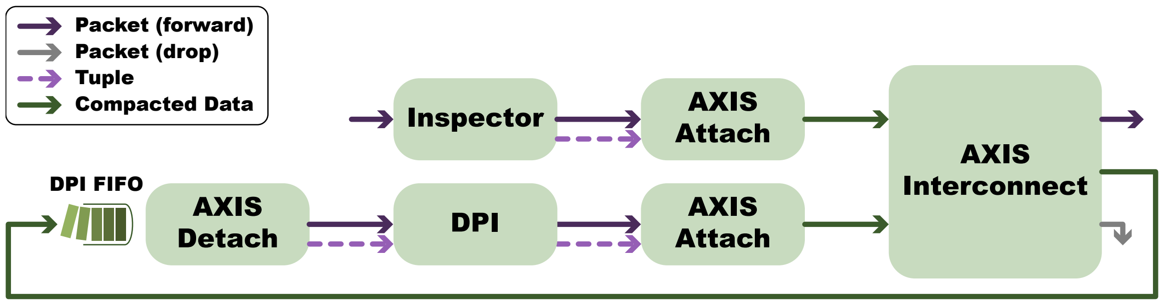

Figure 5.4: DPI Filter Hardware Implementation Scheme.

The flow of packets and tuples among the components is fully implemented using AXI4-Stream interfaces.

Inspector Compiled SDNet module that identifies the packet payload location and length, forwarding packets and their corresponding metadata tuple through two different AXI4-Stream interfaces (due to SDNet behavior).

DPI Compiled SDNet module that updates the ASCII burst and proportion values from the input tuple and computes the decision for the associated packet: "plain", "encrypted" or "needs further recirculation".

AXIS Attach HDL module that attaches a tuple as the TUSER signal and a decision flag as the TDEST signal of the corresponding packet, producing a single AXI4-Stream output port which carries all the compacted data: packet, tuple and decision.

AXIS Detach HDL module that conversely dettaches the TUSER and TDEST signals from its input port into separate tuple and decision AXI4-Stream interfaces.

AXIS Interconnect AXI4-Stream routing module in charge of handling the compacted packets it receives to the appropriate output according to the TDEST (decision) signal, namely: forward data to the *DPI* module, forward data to the system output port, or discard data (using an unrouted value of TDEST).

DPI FIFO Hardware FIFO holding packets that the *AXIS Interconnect* has forwarded to the *DPI* module, a convenient length can be determined from traffic analysis by the statistical methods exposed during the previous section.

This architecture was implemented using Vivado and experiments with similar context to the TLS Client Hello and DNS filters discussed in previous sections were run.

5.5. Results

In order to evaluate the proper functioning of the filter with realistic network traffic, the system was tested under different networks, including that of the teaching laboratories of the university –wired using Gigabit Ethernet–.

Several conclusions are worth mentioning from the outcome of the experiments:

- The DPI filter eventually classified all packets correctly between plain and encrypted. The output packets (plain traffic) were compared to those of the original HDL filter implementation [19] and the results were identical. The thresholds provided for testing were ASCII bursts of 12 consecutive characters and a minimum of 50% of the payload bytes within printable ASCII range.

- Packet order was altered since the feedback policies cause small packets to spend much less time in the system than larger packets that need a huge number of recirculations. Hence, small packets are quickly forwarded while large ones suffer considerable delay as they go through the processing and queuing units.

- Latency metrics were noticeably increased due to the recursion strategy of the *DPI* SDNet module implementation. Sequentially parsing each byte up to 63 times causes comparators from the tuple updation logic and internal control structures of SDNet as well as their respective signal paths to trigger too often. As a consequence, a single pass of the *DPI* module takes 784 clock cycles to complete, which then gets multiplied by the number of recirculations required, causing really large latencies for very large packets (e.g.: 1 KB packets take over 10,000 cycles).

All in all, we observe that SDNet solutions struggle slightly when executing the most complex filters. But at the same time, we prove that it is still possible and relatively simple to get them working using PX with a few tweaks in the design architecture.

6 Remarks

The goal of this project has been analyzing the benefits of application-specific high-level languages for the development of Data Plane networking applications, especially focusing on packet filters for network traffic monitoring. The main question to answer is whether the non-recurring expenses could be drastically reduced by the productivity and efficiency boosts provided by state-of-the-art software suites.

More specifically, the study compares Xilinx SDNet against highly-optimized HDL code. Results obtained with SDNet are positive: the total number of lines of code has been divided by a factor of 5 in the worst case, whilst latency is merely increased by no more than 270 ns (83 clock cycles). Moreover, the SDNet solution is perfectly able to cope with a fully saturated link at 100 Gbps. Although the main drawback of the SDNet solutions is the relatively high resource usage, it actually does not surpass 2.5% of the total capacity of the FPGA available in the Xilinx VCU108 Evaluation Kit.

Additionally, a more complex DPI filter was also implemented using SDNet. Some limitations currently present in the PX language were aired when developing the system, but they could be circumvented by increasing the complexity level in the main architecture. While these changes allow for a correct hardware implementation of the filter, the resulting solution inevitably incurs into issues like unrealistic latencies and the introduction of an input FIFO that must be properly dimensioned using stochastic mechanisms like queuing models. This is merely a symptom of the original idea of SDNet as a network routing –and not filtering– development environment, although we have clearly seen the productivity and potential this tool can bring to the latter scenario. In any case, SDNet is still in a rather early stage of its life cycle, so such potential could greatly improve during the upcoming years, even to the point of surpassing said limitations and turning into an utterly powerful packet processor.

Our last conclusion is that Xilinx SDNet can be a very valuable tool for the development of network packet filters. Though we found certain restraints and disadvantages in terms of resource utilization and latency, the benefits in terms of productivity, code maintainability and time-to-market, as well as the ability of designs to operate at line rate at 100 Gbps, might overshadow these disadvantages.

7 Future Work

Throughout the development of this project, several thoughts have emerged about further investigation with the different tools and designs that have been discussed, exploring ideas in terms of improvability and extensibility:

- Export the abstraction level that SDNet brings to the whole Vivado Design Flow. Ideally, the end user would only have to choose the target board and interfaces involved, and then write a PX program describing the desired behavior of the Data Plane, leaving all the HDL translation, synthesis, implementation, routing, optimization, constraining and FPGA programming work to the automated engine of the development environment. That way PX could turn into the the perfect time-to-market solution that P4 originally wanted manufacturers to create.

- Study hybrid software-hardware designs that connect SDNet filters with a CPU that makes simple estimations of certain parameters of the filter –like the one for the input FIFO of the DPI filter–, which could then be fed back to the filter enabling real-time adaptability. This is specially interesting if we employ a ping-pong technique for partial reconfiguration of the filter element that needs to be updated.

- The performance benchmarks that were ran during this work are only a proportion of the myriad test scenarios where SDNet could prove useful. Many of the current hardware problems from the different fields of network computing could be easily implemented using PX and compared against the state of the art alternatives in order to evaluate the quality of the different solutions against the development time. For example, creating a high-performance hardware firewall application using SDNet could pose an interesting challenge and end up rising promising results.
- As already stated earlier in this document, some design decisions relative to the SDNet compiler are also a matter of concern for the future of the tool, since they restrict the capabilities of the PX language for modeling more complex packet flows in the Data Plane. As it happens, the way state machines are inferred and instantiated could be reworked to allow deeper protocol recursivity and greatly improve the flexibility of SDNet.

In other words, this work marks an investigation line in packet filtering for SDNet, but countless applications and solutions are yet to be discovered with the main objective of relieving the pressure on the actual coding process of a project, so that the focus can be set on new and creative ideas which then can become real with relatively little effort.

A
Formal Proof of Little's Law

Little's Law states that the long-term average number L of items in a stationary system is equal to the long-term average effective arrival rate λ multiplied by the average time W that an item spends in the system. That is:

$$L = \lambda W$$

The following straightforward proof approach is inspired by an utterly recent work that was published in 2011 on the ocassion of the 50th Anniversary of Little's Law [21].

A.1. Proof of Little's Law for a System Empty at 0 and T

First, consider an scenario of a queuing process over a time interval $[0, T]$. Let:

$n(t) :=$ number of items in the system at time t.

$\lambda :=$ average arrival rate in $[0, T]$ (items/time unit).

$N :=$ number of items arriving in $[0, T]$.

$L :=$ average number of items in the system during $[0, T]$.

$W :=$ average waiting time of an item during $[0, T]$ (time units).

$A = \int_0^T n(t)\, dt :=$ area under $n(t)$ over $[0, T]$ (time units).

Theorem A.1 (Little's Law). *For a queuing system observed over* $[0, T]$ *that is empty at* 0 *and* T *and has* $0 < T < \infty$, *the formula* $L = \lambda W$ *holds.*

Proof. Using the notation in the left-hand column above, we see that:

$$L = \tfrac{A}{T}$$
$$\lambda = \tfrac{N}{T}$$
$$W = \tfrac{A}{N}$$

Whence:

$$L = \frac{A}{T} = \frac{A}{T} \cdot \frac{N}{N} = \frac{N}{T} \cdot \frac{A}{N} = \lambda W$$

□

A.2. Proof of Little's Law with Permissible Initial and Final Queues in $[0, T]$

We can now establish a more general result by supressing the restriction of the empty queues at times 0 and T.

Theorem A.2 (Little's Law over $[0, T]$). *For a queuing system observed over* $[0, T]$ *that has* $0 < T < \infty$, *the formula* $L = \lambda W$ *holds.*

Proof. In Theorem A.1 we defined N as the total arrivals in $[0, T]$. Here we introduce:

$S(t) :=$ cumulative number of items in the system over $[0, t]$.

This includes not only the cumulative arrivals up to t, but also any items that were in the system at $t = 0$. This permits $S(0) = n(0) > 0$ and $n(T) > 0$, in contrast to A.1.

The definition of A continues the same as before. Otherwise, paralleling the arguments used to prove Theorem A.1 above, we obtain:

$$L = \tfrac{A}{T}$$
$$\lambda = \tfrac{S(T)}{T}$$
$$W = \tfrac{A}{S(T)}$$

Whence:

$$L = \frac{A}{T} = \frac{A}{T} \cdot \frac{S(t)}{S(t)} = \frac{S(t)}{T} \cdot \frac{A}{S(t)} = \lambda W$$

□

B Kolmogorov-Smirnov Test

The following explanation of the Kolmogorov-Smirnov test is based in the lecture notes of the Statistics II course conducted by Amparo Baíllo Moreno in 2017.

Let $X_1, \dots, X_n$ be a random sample of an stochastic variable $X \sim F$.

We state the contrast:

$$\text{Null Hypothesis } H_0 : F = F_0$$

Where $F_0 \in \mathcal{C}^0$ is a continuous CDF completely specified.

For such purpose, we define the ECDF as follows:

$$F_n(x) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}_{\{X_i \leq x\}}$$

Here, $\mathbb{1}_A$ represents the characteristic function of the set A:

$$\mathbb{1}_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{otherwise} \end{cases}$$

Theorem B.1 (Glivenko-Cantelli)**.**

$$D_n = \|F_n - F\|_\infty = \sup_{x \in \mathbb{R}} |F_n(x) - F(x)| \xrightarrow[n \to \infty]{\text{C.S.}} 0$$

The idea behind the contrast of the Kolmogorov-Smirnov test is to reject H_0 inside the rejection region given by $R = \{D_n > C_\alpha\}$ for an appropriate critical value of C_α.

Lemma B.1. *If a stochastic variable X has a continuous CDF F, then $F(X)$ has uniform distribution in $(0,1)$.*

Theorem B.2. *Under the null hypothesis H_0, the distribution of D_n is identical for any possible continuous CDF F_0.*

Proof. Using the properties of F as a CDF, with a probability equal to 1:

$$D_n = \sup_{x\in\mathbb{R}} \left| \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}_{\{F(X_i)\leq F(x)\}} - F(x) \right| = \sup_{u\in[0,1]} \left| \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}_{\{U_i \leq u\}} - u \right|$$

Where:

$$U_i = F(X_i) \sim \text{Unif}\,[0,1]$$

Or alternatively:

$$D_n = \max\{D_n^+, D_n^-\}$$

Where:

$$D_n^+ := \sup_{x\in\mathbb{R}} [F_n(x) - F(x)] = \max_{1\leq i\leq n} \left[\tfrac{i}{n} - F(X_i)\right]$$
$$D_n^- := \sup_{x\in\mathbb{R}} [F(x) - F_n(x)] = \max_{1\leq i\leq n} \left[F(X_i) - \tfrac{i-1}{n}\right]$$

□

As a consequence, the value of C_α for the rejection region $R = \{D_n > C_\alpha\}$ is the same for any continuous CDF F_0.

The CDF of D_n itself has a clossed expression and can also be easily simulated with arbitrary precision for later tabulation.

C PX Source Code of Implemented SDNet Filters

C.1. TLS Client Hello Filter

Listing C.1: PX Source Code for ClientHello Module

```
// Interface subclasses
class PktIn   :: Packet(in)  {}
class PktOut  :: Packet(out) {}
class MetadataIn :: Tuple(in) {
    struct {
        payload_exists     :  1, // Whether payload is present
        payload_length     : 32, // Payload length in bits
        payload_offset     : 32, // Payload offset in bits
        is_client_hello    :  1  // Whether packet is client hello
    }
}
class MetadataOut :: Tuple(out) {
    struct {
        payload_exists     :  1, // Whether payload is present
        payload_length     : 32, // Payload length in bits
        payload_offset     : 32, // Payload offset in bits
        is_client_hello   :  1  // Whether packet is client hello
    }
}

class ClientHello :: System {
    PktIn     instream;
    PktOut    outstream;
    ClientHello_Parser parser;
    ClientHello_Editor editor;
    method connect = {
        parser.packet_in = instream,
        editor.packet_in = parser.packet_out,
        editor.tuple_in  = parser.tuple_out,
        outstream = editor.packet_out
    }
} // ClientHello

class ClientHello_Parser :: ParsingEngine(12000, 4, ETH) {

    // Constants
    // const VLAN_TYPE = 0x8100;
    const IPV4_TYPE = 0x0800;
    // const IPV6_TYPE = 0x86dd;
    const TCP_TYPE = 0x06;
    // const UDP_TYPE = 0x11;
    const SUCCESS = 0;
    const FAILURE = 1;
    MetadataOut tuple_out;

    // Ethernet MAC header
    class ETH :: Section(1) {
        struct {
            skip : 96, // Skip fields
            type : 16  // Tag Protocol Identifier
        }
        // ETH can be followed by VLAN, IPV4 or IPV6
        map types {
            // (VLAN_TYPE, VLAN),
            (IPV4_TYPE, IPV4),
            // (IPV6_TYPE, IPV6),
            done(SUCCESS)
        }
        // Initialise tuple
        method update = {
            tuple_out.payload_exists  = 0,
            tuple_out.payload_length  = 0,
            tuple_out.payload_offset  = sizeof(ETH),
            tuple_out.is_client_hello = 0
        }
        // Identify following protocol
        method move_to_section = types(type);
        // Move to following protocol
        method increment_offset = sizeof(ETH);
    } // ETH

    // IPV4 header
    class IPV4 :: Section(2) {
        struct {
            version : 4,  // Version (4)
            hdrlen  : 4,  // Header Length
            tos     : 8,  // Type of Service
            length  : 16, // Total Length
            skip    : 40, // Skip fields
            proto   : 8   // Next Protocol
        }
        method update = {
            // Save payload length
            tuple_out.payload_length = (8 * length) - (32 * hdrlen),
            // Update payload offset
            tuple_out.payload_offset = tuple_out.payload_offset + (32 * hdrlen)
        }
        // IPV4 can be followed by TCP
        map types {
            (TCP_TYPE, TCP),
            // (UDP_TYPE, UDP),
            done(SUCCESS)
        }
        // Identify following protocol
        method move_to_section = types(proto);
        // Move to following protocol
        method increment_offset = hdrlen * 32;
    } // IPV4

    // TCP header
    class TCP :: Section(3) {
        struct {
            skip    : 96, // Skip fields
            dataoff : 4   // Data Offset
        }
        method update = {
            // Mark payload as present
            tuple_out.payload_exists = 1,
            // Update payload length
            tuple_out.payload_length = tuple_out.payload_length - (32 * dataoff),
            // Update payload offset
            tuple_out.payload_offset = tuple_out.payload_offset + (32 * dataoff)
        }
        // Identify following protocol
        method move_to_section =
            if(tuple_out.payload_length - (32 * dataoff) >= sizeof(SSL_CLIENT_HELLO
                )) SSL_CLIENT_HELLO
            else done(SUCCESS);
        // Move to following protocol
        method increment_offset = dataoff * 32;
    } // TCP

    // SSL Client Hello
    class SSL_CLIENT_HELLO :: Section (4) {
        struct {
            rectype : 8,  // Record Content Type
            skip    : 32, // Skip fields
            hstype  : 8   // Handshake Type
        }
        // Flag as Client Hello
        method update = {
            tuple_out.is_client_hello = (rectype == 0x16) && (hstype == 0x01)
        }
        // Identify following protocol
        method move_to_section = done(SUCCESS);
        // Move to following protocol
        method increment_offset = 0;
    } // SSL_CLIENT_HELLO

} // ClientHello_Parser

class ClientHello_Editor :: EditingEngine(12000, 2, FETCH) {
    // Constants
    const SUCCESS = 0;
    const FAILURE = 1;
    MetadataIn tuple_in;

    class FETCH :: Section(1) {
        // Drop only non-ClientHello packets
        method move_to_section =
            if (tuple_in.is_client_hello == 0) DROP
            else done(SUCCESS);
        method increment_offset = 0;
    } // FETCH

    class DROP :: Section(2) {
        // Remove whole packet
        method remove = rop();
        // Finish engine
        method move_to_section = done(SUCCESS);
        method increment_offset = 0;
    } // DROP

} // ClientHello_Editor
```

C.2. DNS Filter

Listing C.2: PX Source Code for DNS Module

```
// Interface subclasses
class PktIn  :: Packet(in)  {}
class PktOut :: Packet(out) {}
class MetadataIn :: Tuple(in) {
    struct {
        is_dns : 1  // Whether packet is DNS
    }
}
class MetadataOut :: Tuple(out) {
    struct {
        is_dns : 1  // Whether packet is DNS
    }
}
```

```
class DNS :: System {
    PktIn     instream;
    PktOut    outstream;
    DNS_Parser parser;
    DNS_Editor editor;
    method connect = {
        parser.packet_in = instream,
        editor.packet_in = parser.packet_out,
        editor.tuple_in  = parser.tuple_out,
        outstream = editor.packet_out
    }
} // DNS

class DNS_Parser :: ParsingEngine(12000, 5, ETH) {

    // Constants
    const VLAN_TYPE = 0x8100;
    const IPV4_TYPE = 0x0800;
    const IPV6_TYPE = 0x86dd;
    // const TCP_TYPE = 0x06;
    const UDP_TYPE = 0x11;
    const SUCCESS = 0;
    const FAILURE = 1;
    MetadataOut tuple_out;

    // Ethernet MAC header
    class ETH :: Section(1) {
        struct {
            skip : 96, // Skip fields
            type : 16  // Tag Protocol Identifier
        }
        // ETH can be followed by VLAN, IPV4 or IPV6
        map types {
            (VLAN_TYPE, VLAN),
            (IPV4_TYPE, IPV4),
            (IPV6_TYPE, IPV6),
            done(SUCCESS)
        }
        // Initialise tuple
        method update = {
            tuple_out.is_dns = 0
        }
        // Identify following protocol
        method move_to_section = types(type);
        // Move to following protocol
        method increment_offset = sizeof(ETH);
    } // ETH

    // VLAN header
    class VLAN :: Section(2:3) {
        struct {
            skip : 16, // Skip fields
            tpid : 16  // Tag Protocol Identifier
        }
        // VLAN can be followed by VLAN, IPV4 or IPV6
        map types {
            (VLAN_TYPE, VLAN),
            (IPV4_TYPE, IPV4),
            (IPV6_TYPE, IPV6),
            done(SUCCESS)
        }
        // Identify following protocol
        method move_to_section = types(tpid);
        // Move to following protocol
        method increment_offset = sizeof(VLAN);
    } // VLAN

    // IPV4 header
    class IPV4 :: Section(2:4) {
        struct {
            version : 4,  // Version (4)
            hdrlen  : 4,  // Header Length
            skip    : 64, // Skip fields
            proto   : 8   // Next Protocol
        }
        // IPV4 can be followed by TCP
        map types {
            (UDP_TYPE, UDP),
            done(SUCCESS)
        }
        // Identify following protocol
        method move_to_section = types(proto);
        // Move to following protocol
        method increment_offset = hdrlen * 32;
    } // IPV4

    // IPV6 header
    class IPV6 :: Section(2:4) {
        struct {
            skip    : 48, // Skip fields
            nexthdr : 8   // Next Header
        }
        // IPV4 can be followed by TCP
        map types {
            (UDP_TYPE, UDP),
            done(SUCCESS)
        }
        // Identify following protocol
        method move_to_section = types(nexthdr);
        // Move to following protocol
        method increment_offset = 320;
    }

    // UDP header
    class UDP :: Section(3:5) {
        struct {
            srcport : 16, // Source Port
            dstport : 16  // Destination Port
        }
        // Flag as DNS
        method update = {
            tuple_out.is_dns = (srcport == 53) || (dstport == 53)
        }
        // Identify following protocol
        method move_to_section = done(SUCCESS);
        // Move to following protocol
        method increment_offset = 0;
    } // UDP

} // DNS_Parser

class DNS_Editor :: EditingEngine(12000, 2, FETCH) {

    // Constants
    const SUCCESS = 0;
    const FAILURE = 1;
    MetadataIn tuple_in;

    class FETCH :: Section(1) {
        // Drop only non-DNS packets
        method move_to_section =
            if (tuple_in.is_dns == 0) DROP
            else done(SUCCESS);
        method increment_offset = 0;
    } // FETCH

    class DROP :: Section(2) {
        // Remove whole packet
        method remove = rop();
        // Finish engine
        method move_to_section = done(SUCCESS);
        method increment_offset = 0;
    } // DROP

} // DNS_Editor
```

C.3. DPI Filter

Listing C.3: PX Source Code for Inspector Module

```
// Interface subclasses
class PktIn  :: Packet(in)  {}
class PktOut :: Packet(out) {}
class TplOut :: Tuple(out) {
    struct {
        payload_offset    : 32, // Payload offset in bits
        payload_length    : 32  // Payload length in bits
    }
}

class Inspector :: System {
    PktIn  instream;
    PktOut outstream;
    TplOut tuple_out;
    Inspector_Parser parser;
    method connect = {
        parser.packet_in = instream,
        outstream = parser.packet_out,
        tuple_out = parser.tuple_out
    }
} // Inspector

class Inspector_Parser :: ParsingEngine(12000, 64, ETH) {

    // Constants
    const MPLS_UNI_TYPE = 0x8847;
    const MPLS_MUL_TYPE = 0x8848;
    const VLAN_TYPE = 0x8100;
    const IPV4_TYPE = 0x0800;
    const IPV6_TYPE = 0x86dd;
    const TCP_TYPE = 0x06;
    const UDP_TYPE = 0x11;
    const SUCCESS = 0;
    const FAILURE = 1;
    TplOut tuple_out;

    // Ethernet header
    class ETH :: Section(1) {
        struct {
            skip : 96, // Skip fields
            type : 16  // Tag Protocol Identifier
        }
        // Mapping for next headers
        map types {
            (MPLS_UNI_TYPE, MPLS),
            (MPLS_MUL_TYPE, MPLS),
            (VLAN_TYPE, VLAN),
            (IPV4_TYPE, IPV4),
            (IPV6_TYPE, IPV6),
            done(SUCCESS)
        }
        // Update output tuple
        method update = {
            tuple_out.payload_offset = sizeof(ETH),
            tuple_out.payload_length = 0
        }
        // Next header lookup
        method move_to_section = types(type);
        // Current header skip
        method increment_offset = sizeof(ETH);
    } // ETH

    // MPLS header
    class MPLS :: Section(2:64) {
        struct {
            skip : 23, // Skip fields
            bos  : 1   // Bottom Of Stack
        }
        // Mapping for next headers
        map types {
            (0, MPLS),
            (1, IPVX),
            done(SUCCESS)
        }
        // Update output tuple
        method update = {
            tuple_out.payload_offset = tuple_out.payload_offset + 32
        }
        // Next header lookup
        method move_to_section = types(bos);
        // Current header skip
        method increment_offset = 32;
    } // MPLS

    // VLAN header
    class VLAN :: Section(2:64) {
        struct {
            skip : 16, // Skip fields
            tpid : 16  // Tag Protocol Identifier
        }
        // Mapping for next headers
        map types {
            (MPLS_UNI_TYPE, MPLS),
            (MPLS_MUL_TYPE, MPLS),
            (VLAN_TYPE, VLAN),
            (IPV4_TYPE, IPV4),
            (IPV6_TYPE, IPV6),
            done(SUCCESS)
        }
        // Update output tuple
        method update = {
            tuple_out.payload_offset = tuple_out.payload_offset + sizeof(VLAN)
        }
        // Next header lookup
        method move_to_section = types(tpid);
        // Current header skip
        method increment_offset = sizeof(VLAN);
    } // VLAN

    // IPV4/IPV6 discriminating header
    class IPVX :: Section(2:64) {
        struct {
            version : 4 // Version
        }
        // Mapping for next headers
        map types {
            (0x4, IPV4),
            (0x6, IPV6),
            done(SUCCESS)
        }
        // Next header lookup
        method move_to_section = types(version);
        // Current header skip
        method increment_offset = 0;
    } // IPVX

    // IPV4 header
    class IPV4 :: Section(2:64) {
        struct {
            version : 4,  // Version (4)
            hdrlen  : 4,  // Header Length
            tos     : 8,  // Type of Service
            length  : 16, // Total Length
            skip    : 40, // Skip fields
            proto   : 8   // Next Protocol
        }
        // Mapping for next headers
        map types {
            (TCP_TYPE, TCP),
            (UDP_TYPE, UDP),
            done(SUCCESS)
        }
        // Update output tuple
        method update = {
            tuple_out.payload_offset = tuple_out.payload_offset + (32 * hdrlen),
            tuple_out.payload_length = (8 * length) - (32 * hdrlen)
        }
        // Next header lookup
        method move_to_section = types(proto);
        // Current header skip
        method increment_offset = 32 * hdrlen;
    } // IPV4

    // IPV6 header
    class IPV6 :: Section(1:64) {
        struct {
            skip    : 32, // Skip fields
            length  : 16, // Payload Length
            nexthdr : 8   // Next Header
        }
        // Mapping for next headers
        map types {
            (TCP_TYPE, TCP),
            (UDP_TYPE, UDP),
            done(SUCCESS)
        }
        // Update output tuple
        method update = {
            tuple_out.payload_offset = tuple_out.payload_offset + 320,
            tuple_out.payload_length = 8 * length
        }
        // Next header lookup
        method move_to_section = types(nexthdr);
        // Current header skip
        method increment_offset = 320;
    } // IPV6

    // TCP header
    class TCP :: Section(1:64) {
        struct {
            skip    : 96, // Skip fields
            dataoff : 4   // Data Offset
        }
        // Update output tuple
        method update = {
            tuple_out.payload_offset = tuple_out.payload_offset + (32 * dataoff),
            tuple_out.payload_length = tuple_out.payload_length - (32 * dataoff)
        }
        // Next header lookup
        method move_to_section = done(SUCCESS);
        // Current header skip
        method increment_offset = 0;
    } // TCP

    // UDP header
    class UDP :: Section(1:64) {
        // Update output tuple
        method update = {
            tuple_out.payload_offset = tuple_out.payload_offset + 64,
            tuple_out.payload_length = tuple_out.payload_length - 64
        }
        // Next header lookup
        method move_to_section = done(SUCCESS);
        // Current header skip
        method increment_offset = 0;
    } // UDP

} // Inspector_Editor
```

Listing C.4: PX Source Code for DPI Module

```
// Interface subclasses
class PktIn  :: Packet(in)  {}
class PktOut :: Packet(out) {}
class TplInt :: Tuple {
    struct {
        count : 6, // Current parsed byte count
        burst : 6  // Current consecutive ASCII bytes
    }
}
class TplIn :: Tuple(in) {
    struct {
        payload_offset : 32, // Payload offset in bits
        payload_length : 32, // Payload length in bits
        payload_parsed : 32, // Parsed payload in bits
        ascii_count    : 16, // ASCII byte count
        ascii_burst    : 16  // Maximum consecutive ASCII bytes
    }
}
class TplOut :: Tuple(out) {
    struct {
        payload_offset : 32, // Payload offset in bits
        payload_length : 32, // Payload length in bits
        payload_parsed : 32, // Parsed payload in bits
        ascii_count    : 16, // ASCII byte count
        ascii_burst    : 16  // Maximum consecutive ASCII bytes
    }
}
class Decision :: Tuple(out) {
    struct {
        decision       :  8  // Decision: 00 -> Not ASCII
                             //           01 -> ASCII
                             //           11 -> Recirculate
    }
}

class DPI :: System {
    PktIn    instream;
    PktOut   outstream;
    TplIn    tuple_in;
    TplOut   tuple_out;
    Decision decision;
    DPI_Parser parser;
    method connect = {
        parser.packet_in = instream,
        parser.tuple_in  = tuple_in,
        outstream = parser.packet_out,
        tuple_out = parser.tuple_out,
        decision  = parser.decision
    }
} // DPI

class DPI_Parser :: ParsingEngine(12000, 64, SEEK) {

    // Constants
    const SHIFT_TRIGGER = 1;
    const BURST_TRIGGER = 12;
    const SUCCESS = 0;
    const FAILURE = 1;
```

```
TplInt tuple_int;
TplIn  tuple_in;
TplOut tuple_out;
Decision decision;

// SEEK already parsed data
class SEEK :: Section(1) {
    // Initialise tuples
    method update = {
        tuple_out.payload_offset = tuple_in.payload_offset,
        tuple_out.payload_length = tuple_in.payload_length,
        tuple_out.payload_parsed = tuple_in.payload_parsed,
        tuple_out.ascii_count    = tuple_in.ascii_count,
        tuple_out.ascii_burst    = tuple_in.ascii_burst,
        decision.decision        =
            // Mark as ASCII if parameterizable conditions are met
            if ((tuple_in.payload_length > 0)
                && (((tuple_in.ascii_count << 3) >= (tuple_in.payload_length >>
                     SHIFT_TRIGGER))
                || (tuple_in.ascii_burst >= BURST_TRIGGER))) 1
            // Otherwise, mark as not ASCII if completely parsed
            else if (tuple_in.payload_parsed >= tuple_in.payload_length) 0
            // In any other case, mark for recirculation
            else 3,
        tuple_int.count          = 0,
        tuple_int.burst          = 0
    }
    // Identify following protocol
    method move_to_section =
        // Terminate the engine if decision already taken
        if (((tuple_in.ascii_count << 3) >= (tuple_in.payload_length >>
            SHIFT_TRIGGER))
                || (tuple_in.ascii_burst >= BURST_TRIGGER)
                || (tuple_in.payload_parsed >= tuple_in.payload_length)) done(
                    SUCCESS)
        // Continue to byte parser if marked for recirculation
        else BYTE;
    // Move to following protocol
    method increment_offset = tuple_in.payload_offset + tuple_in.payload_parsed
        ;
} // SEEK
```

```
    // Payload byte
    class BYTE :: Section(2:64) {
        struct {
            byte : 8 // Isolated payload byte
        }
        method update = {
            // Update parsed payload count
            tuple_out.payload_parsed = tuple_out.payload_parsed + 8,
            // Update ASCII byte count
            tuple_out.ascii_count = tuple_out.ascii_count + ((byte >= 32) && (byte
                <= 126)),
            // Update current burst count
            tuple_int.burst =
                if ((byte >= 32) && (byte <= 126)) tuple_int.burst + 1
                else 0,
            // Update global burst count
            tuple_out.ascii_burst =
                if ((byte >= 32) && (byte <= 126) && ((tuple_int.burst + 1) >
                    tuple_out.ascii_burst)) tuple_int.burst + 1
                else tuple_out.ascii_burst,
            // Update byte count
            tuple_int.count = tuple_int.count + 1
        }
        // Identify following protocol
        method move_to_section =
            if (((tuple_int.count + 1) < 63) && (tuple_out.payload_parsed + 8 <
                tuple_in.payload_length)) BYTE
            else done(SUCCESS);
        // Move to following protocol
        method increment_offset = sizeof(BYTE);
    } // BYTE

} // DPI_Parser
```

Bibliography

[1] DPDK. Data Plane Development Kit. https://dpdk.org/.

[2] W. Wu, P. DeMar, and M. Crawford. Why Can Some Advanced Ethernet NICs Cause Packet Reordering? *IEEE Communications Letters*, 15(2):253–255, February 2011.

[3] Xilinx. Developer Zone: Design Tools. https://www.xilinx.com/products/design-tools.html.

[4] Intel. Intel® HLS Compiler. https://www.altera.com/products/design-software/high-level-design/intel-hls-compiler/overview.html.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM*, 38:69–74, April 2008.

[6] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: programming protocol-independent packet processors. *ACM SIGCOMM*, 44:87–95, July 2014.

[7] G. Brebner. Programmable hardware for software defined networks. In *2015 European Conference on Optical Communication (ECOC)*, Valencia, Spain, 27 September – 1 October 2015.

[8] J. F. Zazo and S. López-Buedo and G. Sutter and J. Aracil. Automated synthesis of FPGA-based packet filters for 100 Gbps network monitoring applications. In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, 30 Nov. – 2 Dec. 2016.

[9] P. Benácek, V. Pu, and H. Kubátová. P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Washington, DC, USA, 1 – 2 May 2016.

[10] V. Puš, L. Kekely, and J. Kořenek. Low-latency modular packet header parser for FPGA. In *2012 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Austin, TX, USA, 29 – 30 October 2012.

[11] P. Benáček, V. Puš, J. Kořenek, and M. Kekely. Line rate programmable packet processing in 100Gb networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Ghent, Belgium, 4 – 8 September 2017.

[12] H. Wang, R. Soulé, H. Tu Dang, K. Suh Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. P4FPGA: A Rapid Prototyping Framework for P4. In *2017 ACM Symposium on SDN Research (SOSR)*, Santa Clara, CA, USA, 3 – 4 April 2017.

[13] Bluespec. BSV High-Level HDL. http://bluespec.com/54621-2/.

[14] H. Tu Dang, H. Wang, T. Jepsen, G. Brebner, C. Kim, J. Rexford, R. Soulé, and H. Weatherspoon. Whippersnapper: A P4 Language Benchmark Suite. In *2017 ACM Symposium on SDN Research (SOSR)*, Santa Clara, CA, USA, 3 – 4 April 2017.

[15] A. Fiessler, S. Hager, B. Scheuermann, and A. W. Moore. HyPaFilter – A versatile hybrid FPGA packet filter. In *2016 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Santa Clara, CA, USA, 17 – 18 March 2016.

[16] S. Hager, D. Bendyk, and B. Scheuermann. Matching circuits can be small: Partial evaluation and reconfiguration for FPGA-based packet processing. *Journal of Parallel and Distributed Computing (JPDC)*, 109:42–49, November 2017.

[17] N. Zilberman, Y. Audzevich, G. Adam Covington, and A. W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 34:32–41, September – October 2014.

[18] Xilinx. Xilinx Virtex UltraScale FPGA VCU108 Evaluation Board. https://www.xilinx.com/products/boards-and-kits/ek-u1-vcu108-g.html.

[19] M. Ruiz, G. Sutter, S. López-Buedo, and J. E. López de Vergara. FPGA-based encrypted network traffic identification at 100 Gbit/s. In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, 30 November – 2 December 2016.

[20] V. Paxson and S. Floyd. Wide area traffic: the failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3:226–244, June 1995.

[21] D. Simchi-Levi and M. A. Trick. Little's Law as Viewed on Its 50th Anniversary. *Operations Research*, 59:535, May 2011.