KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Embedded Systems

Second cycle, 30 credits

FPGA accelerated packet capture with eBPF

Performance considerations of using SoC FPGA accelerators for packet capturing.

JAKUB DUCHNIEWICZ

Stockholm, Sweden, 2022

FPGA accelerated packet capture with eBPF

Performance considerations of using SoC FPGA accelerators for packet capturing.

JAKUB DUCHNIEWICZ

Master's Programme, ICT Innovation, 120 credits
Date: December 9, 2022

Supervisors: Sebastian Pisklak, Hasini Thilanka Thilakasiri Laddusinghe Badu
Examiner: Matthias Becker
School of Electrical Engineering and Computer Science
Host company: Tietoevry
Swedish title: FPGA-accelererad paketfångst med eBPF
Swedish subtitle: Prestandaöverväganden vid användning av SoC FPGA acceleratorer för paketering.

Abstract

With the rise of the Internet of Things and the proliferation of embedded devices equipped with an accelerator arose a need for efficient resource utilization. Hardware acceleration is a complex topic that requires specialized domain knowledge about the platform and different trade-offs that have to be made, especially in the area of power consumption. Efficient work offloading strives to reduce or at least maintain the total power consumption of the system. Offloading packet capturing is usually done in more powerful devices, hence scarce research is present concerning network packet acceleration in embedded devices.

The thesis focuses on accelerating networking packets utilizing a Field Programmable Gate Array in an embedded Linux System. The solution is based on a custom Linux distribution assembled using the Buildroot tool, specially configured and patched Linux kernel, uboot bootloader, and the programmable logic for packet acceleration. The system is evaluated on a De0-Nano System on Chip development board through modifications to burst lengths, packet sizes, and programmable logic clock frequency. Metrics include packet capturing time, time per packet, and consumed power. Finally, the results are contrasted with baseline embedded Linux packet processing by inspection of a packet's path through the kernel.

Collected results provide a deeper understanding of the packet acceleration problem in embedded devices and the resultant system gives a solid starting point for possible extensions such as packet filtering. Key findings include an improvement in packet processing speed as the clock frequency and burst length are increased while maintaining power consumption. Additionally, the solution performs better when the packet sizes are above 64 bytes as the overhead of additional logic necessary for their processing is compensated. The project is also found to be significantly faster than regular in kernel processing with the caveat of providing just packet capturing whereas Linux contains a full network stack.

Keywords

Field Programmable Gate Array, Acceleration, Networking, Embedded Linux

Sammanfattning

I och med uppkomsten av sakernas internet och spridningen av inbyggda enheter som är utrustade med en accelerator har det uppstått ett behov av effektivt resursutnyttjande. Hårdvaruacceleration är ett komplext ämne som kräver specialiserad domänkunskap om plattformen och olika avvägningar som måste göras, särskilt när det gäller energiförbrukning. Effektiv arbetsavlastning strävar efter att minska eller åtminstone bibehålla systemets totala energiförbrukning. Avlastning av paketering sker vanligtvis i kraftfullare enheter, och därför finns det knappt någon forskning om nätverksacceleration av paket i inbyggda enheter.

Avhandlingen är inriktad på att påskynda nätverkspaket med hjälp av en Field Programmable Gate Array i ett inbäddat Linuxsystem. Lösningen bygger på en anpassad Linuxdistribution som sammanställts med hjälp av verktyget Buildroot, en särskilt konfigurerad och patchad Linuxkärna, uboot bootloader och den programmerbara logiken för paketacceleration. Systemet utvärderas på ett De0-Nano System on Chip-utvecklingskort genom ändringar av burstlängder, paketstorlekar och den programmerbara logikens klockfrekvens. Metrikerna omfattar tid för paketering, tid per paket och förbrukad effekt. Slutligen jämförs resultaten med grundläggande inbäddad Linux-paketbehandling genom inspektion av paketens väg genom kärnan.

De samlade resultaten ger en djupare förståelse för problemet med paketacceleration i inbyggda enheter och det resulterande systemet ger en solid utgångspunkt för möjliga utvidgningar, t.ex. paketfiltrering. Bland de viktigaste resultaten kan nämnas en förbättring av hastigheten i paketbehandlingen när klockfrekvensen och burstlängden ökas samtidigt som strömförbrukningen bibehålls. Dessutom fungerar lösningen bättre när paketstorleken är större än 64 bytes eftersom den extra logik som krävs för att behandla paketen kompenseras. Projektet har också visat sig vara betydligt snabbare än vanlig kärnbearbetning, med den reservationen att det bara tillhandahåller paketupptagning, medan Linux innehåller en fullständig nätverksstack.

Nyckelord

Field Programmable Gate Array, Acceleration, Nätverksarbete, Inbyggd Linux

Streszczenie

Rozwój Internetu Rzeczy i rosnąca popularność systemów wbudowanych posiadających wbudowany akcelerator sprzętowy sprawiły, że wzrosła potrzeba na ich efektywne wykorzytanie. Akceleracja sprzętowa jest dziedziną nauki, która wymaga specjalistycznej wiedzy na temat platformy na której ma operować oraz wymaga znajomości potencjalnych komplikacji które się z nią wiążą. Efektywna akceleracja ma na celu redukcję zużycia energii, a przynajmnniej jej utrzymanie na dotychczasowym poziomie. Tematyka ta jest dość uboga pod kątem dostępnej literatury, gdyż zazwyczaj akceleratory stosowane do sieciowych rozwiązań są używane w rozwiązaniach serwerowych gdzie występują innego rodzaju problemy.

W pracy wykorzystany jest akcelerator Field Programmable Gate Array który jest częścią płytki deweloperskiej De0-Nano System on Chip, gdzie działa współpracując z wbudowanym systemem Linux, do którego przygotowania wykorzystano narzędzie Buildroot. Na końcowe rozwiązanie ponadto składa się połatane jądro Linuxa, bootloader uboot oraz programowalna logika realizująca przechwytywanie pakietów sieciowych. Rozwiązanie poddane jest testom, w których parametry odpowiedzialne za długość transakcji typu burst, rozmiaru pakietu oraz częstotliwości zegara są poddawane modyfikacjom. Wyniki są przedstawione za pomocą czasu przetwarzania pakietu, czasu per pakiet oraz zużycia mocy. Do oceny efektywności rozwiązania posłużyło także porównanie z czasem procesowania pakietu w niezmodyfikowanym systemie Linux

Na podstawie eksperymentów dokonanych w pracy wysunięte są następujące wnioski: wraz ze wzrostem częstotliwości zegara oraz długości transakcji burst, czas procesowania pakietów maleje a zużycie prądu pozostaje na dotychczasowym poziomie. Pakiety o rozmiarze przekraczającym 64 bajty są procesowane wydajniej w dostarczonym rozwiązaniu poprzez kompensację dodatkowego nakładu czasu narzuconego przez logikę zarządzającą przetwarzaniem. System porównano także do zwykłego przetwarzania pakietów odbywającego się w systemie Linux które okazało się zdecydowanie wolniejsze z zastrzeżeniem, iż ów system dokonuje pełnego przetworzenia pakietów a rozwiązanie w pracy jedynie ich przechwytywania. Projekt stanowi podstawę do ewentualnych rozszerzeń, na przykład filtrowania pakietów. Wnioski wysunięte służą pogłębieniu wiedzy w domenie sieci wbudowanych systemów Linux oraz sprzętowej akceleracji.

Słowa kluczowe

Field Programmable Gate Array, sprzętowa akceleracja, sieci internetowe, wbudowany system Linux

Acknowledgments

I would like to thank Sebastian for his insurmountable support regarding the FPGA part of the system and with providing me with his guidance in numerous difficult concepts related to digital logic design. I would also like to help my other colleagues from Tietoevry who were genuinely interested in my thesis and provided valuable input regarding problems I faced during its development.

Additionally, I would like to thank for the help and guidance I received from my examiner Matthias Becker and my supervisor Hasini Thilanka Thilakasiri Laddusinghe Badu.

Lastly, I would like to thank my family for their unyielding enthusiasm and genuine interest in my research. Special appreciation to my brother Szymon who endured my descriptions of some problems faced during the research.

Stockholm, December 2022
Jakub Duchniewicz

Contents

List of Figures

List of Tables

Listings

List of acronyms and abbreviations

ADC	Analog to Digital Converter
AMBA	Advanced Microcontroller Bus Architecture
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
ARP	Address Resolution Protocol
ARTESSO	Advanced Real Time Embedded Silicon System Operator
ASIC	Application Specific Integrated Circuit
ATM	Automatic Teller Machine
AXI	Advanced eXtensible Interface
BPF	Berkeley Packet Filter
CLI	Command Line Interface
CoAP	Constrained Application Protocol
CUDA	Compute Unified Device Architecture
DHCP	Dynamic Host Configuration Protocol
DL	Deep Learning
DPDK	Data Plane Development Kit
eBPF	extended Berkeley Packet Filter
FCS	Frame Cyclic Sequence
FEC	Forward Error Correction
FIFO	First-In First-Out
FOSS	Free Open Source Software
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPGPU	General Purpose Graphics Processing Unit
HDL	Hardware Description Language
HLS	High Level Synthesis
HPS	Hard Processor System
HTTP	Hypertext Transfer Protocol
HTTPS	Secure Hypertext Transfer Protocol

ICMP	Internet Control Message Protocol
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IP	Intellectual Property
IRQ	Interrupt Request
MCU	Microcontroller Unit
ML	Machine Learning
MQTT	Message Queue Telemetry Transport
MTU	Maximum Transmission Unit
NAPI	New API
NIC	Network Interface Card
NoC	Network on Chip
NPF	Network Packet Filter
PL	Programmable Logic
QoS	Quality of Service
RTL	Register Transfer Language
RTOS	Real-Time Operating System
SoC	System on Chip
SPI	Serial Peripheral Interface
TPU	Tensor Processing Unit
WSN	Wireless Sensor Network
XDP	eXpress Data Path

Chapter 1

Introduction

Ever since Defense Advanced Research Project Agency started its research on the time-sharing of computers, the need for processing packets of data by peers on the net has been rising ceaselessly. Now, more than 80 years after the conception of the Internet we find ourselves surrounded by connected devices and an ever-rising amount of data being transferred, be it via the fiberglass, copper wires, or in the air [1]. Every day, we devise new means of utilizing the Internet in a faster, safer, and less bothersome way. The adoption of the Internet of Things has heavily influenced the volume of traffic and uncovered the need for fast and low-power packet processing on Edge devices [2, 3].

Although regular Network Interface Cards (NICs) are usually equipped with some kind of packet-accelerating silicon, they are usually absent in embedded devices that lack exposure to such high volumes of traffic. Nevertheless, with the adoption of Internet of Thingss (IoTs) and the rise of Edge Computing, the niche of low-power embedded devices equipped with a hardware accelerator arose.

Since creating and debugging Application Specific Integrated Circuits (ASICs) is a time and money-consuming process they are usually developed for big projects and mass use. However, if one has a need for a more versatile device and, most importantly, a cheaper one, they usually pick a board equipped with a Field Programmable Gate Array (FPGA) which is a clear winner in terms of reconfigurability and accessibility to the broader populace.

Once packets are accelerated and if they are not retransmitted or dropped, reach the end user who finally consumes them. They, however, can also be captured for analysis, threat detection, or debugging network issues. An example of software allowing for both capturing and visualization of captured data is the widely-known Wireshark program [4]. Even though packet

capturing process does not necessarily require acceleration, it still requires IO operations for storing vast amounts of data on a hard drive posing a technological challenge [5].

Additionally, efficient packet capturing ensues creating a system capable of both filtering the necessary packets out and at the same time not losing any accuracy in timestamps and packet contents. Usually, it is the responsibility of the Linux kernel to do the adequate processing and timestamping of the packets and then pass them to the userspace. Another program that adds additional processing capabilities apart from ordinary capturing is tcpdump [6]. It allows for the usage of various filters and dumping the files in a myriad of different ways.

This thesis focuses on the area of Telecommunications and Embedded Systems, more specifically - network packet acceleration by means of a FPGA. It additionally builds upon the domain of packet capturing and processing and adds a custom FPGA-based capturing IP, paired with a capturing program capable of producing `.pcap` compliant files.

1.1 Structure of the thesis

First, the relevant background topics are introduced, such as packet processing in the Linux operating system, the extended Berkeley Packet Filtering system, or the networking packet acceleration in hardware. Then, the research problem, its purpose, and its goals are described in more detail. Also, the research methodology is introduced in a subsequent chapter where it elaborates on the process of developing the system, modifying various parameters, and finally collecting and processing the resultant data. It is followed by a description of how we approached the problem and what issues we faced during its solution. A detailed description of various components of the system including the programmable logic, Ethernet device driver modifications, and the `.pcap` program is presented in the succeeding chapter. Finally, the results and their interpretations are shown and discussed. The thesis ends with conclusions and future extensions to the project reflections on the research process and area of study.

1.2 Background

The notion of hardware accelerators has been around for quite a while, be it an ordinary sound card, a graphics card or a specialized cryptographic chip, or even an FPGA. In past, programming these entities required immense domain-specific technical knowledge and was therefore quite expensive [7]. Now with the advent of tools that allow for High level synthesis like Intel High Level Synthesis (HLS) or Vivado HLS, any higher-level programmer can make use of the FPGA without resorting to VHDL or SystemVerilog [8]. These tools, obviously, cannot compete with the prowess of the human brain, but they still are a valuable choice when one does not have the engineering resources to write the Hardware Description Language (HDL) [9, 10].

Networking is a domain where such hardware accelerators are widely applicable since it involves high throughput data transfers, encryption, and multiplexing. NICs manufacturers sometimes offer such capabilities, for example, Netronome offers its flagship Agilio product that offers hardware packet acceleration [11]. As previously mentioned, embedded devices are not so powerful and therefore may need to use heterogeneous hardware accelerators such as FPGAs if such are available. Systems like this are but a scratch on a surface of a complex problem of what to accelerate and when.

In order to better understand the system and its various components refer to figure 1.1.

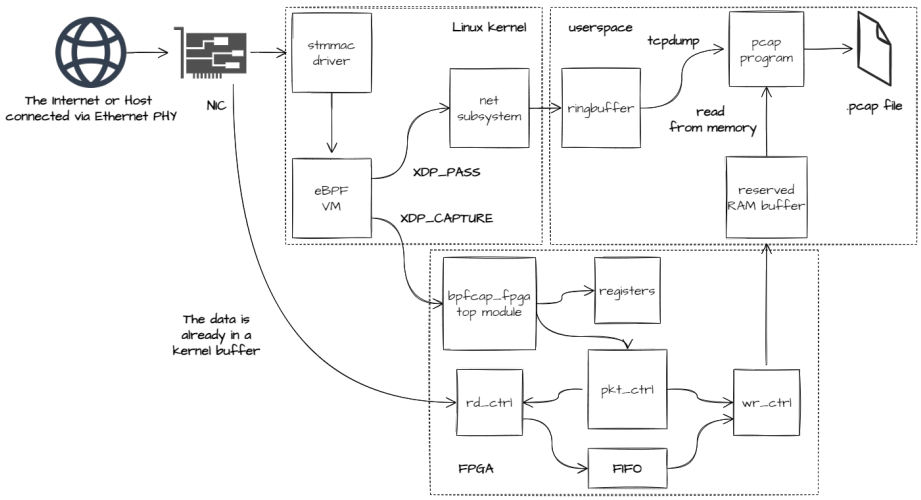


Figure 1.1: High level overview of the system.

As one may see, we have three main components in the project: the kernel part, the FPGA part, and the userspace part. The following subsections will give the necessary background to understand the exact implementations of them in a related chapter 4.

1.2.1 Linux networking overview

The networking in the Linux OS is quite a complex topic and it requires a more detailed description in this subsection. The way of a packet through the Linux kernel is quite complex, riddled with crossroads and conditional junctions, and has several different ways out to the user. Thankfully, the source code is available and a helpful community has created both diagrams and detailed walkthroughs with concrete examples. An example of such a community member is packagecloud.io company, which made two guides, one for ingress and one for egress packets alongside informative graphs. [12, 13, 14]

Perusing any or all of them is very time-consuming so we will instead refer to a simplified diagram of it containing mostly parts relevant to this thesis. In the figure 1.2 you may see that after the packet is received by the NIC it is stored in a ringbuffer structure in RAM, then the Interrupt Request (IRQ) is raised and the CPU starts processing the packet. The softIRQ's handler in this diagram is a function called `stmmac_interrupt`. After this function reads the memory and sees that a packet is indeed there it will call `stmmac_rx` and this function is where most of the New API (NAPI) preparation is performed and where our eXpress Data Path (XDP) function is called as well - `stmmac_xdp_run_prog`. The result of its internal function decides the fate of our packet and as visible in figure 1.2 the packet is either dropped, captured, or passed further via the NAPI function chain to the user.

When a packet is dropped, it is ignored and the memory it occupies can be freed, whereas when a packet is captured it is stored in some special buffer to be later saved to a `.pcap` file. Passing the packet along means the entirety of the Linux *net* subsystem is invoked to process it and then present to the user.

Once the control flow reaches XDP, the next crucial step of processing commences - the extended Berkeley Packet Filter (eBPF) virtual machine execution and a decision on what should be done with the said packet.

1.2.2 eBPF

The eBPF [15, 16, 17] is nothing novel but rather heavily refurbished version of the original Berkeley Packet Filter (BPF). It is constantly developed

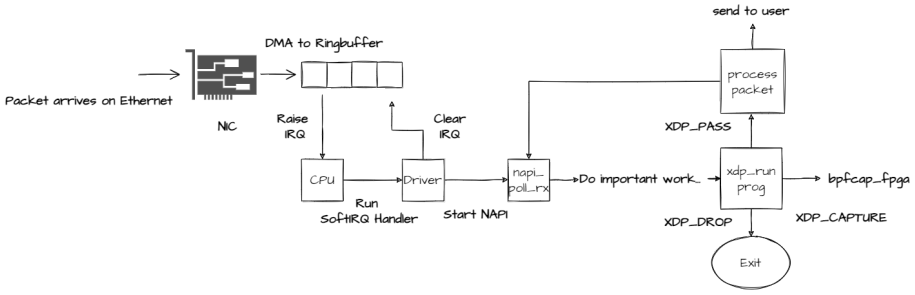


Figure 1.2: Overview of the networking path in the Ethernet driver.

and augmented and its community is steadily growing due to a ceaseless need for Cloud orchestration and efficient and unified resource monitoring on Linux systems [8]. Even though it excels in performance on server Linux distributions where hundreds or thousands of microservices need to be monitored, it has also found some adoption in the embedded world. If one can afford a Linux distribution on their target platform, then they surely should consider enabling eBPF via kernel config.

Though not limited to, it works in conjunction with the Linux networking stack to determine whether the current packet should be dropped, passed further for processing, or transmitted to another interface. In the case of our system, we added an additional option for capturing the packet via the FPGA. The system relies on a virtual machine and optional Just-in-Time compilation that adds additional overhead to our kernel but allows for a much greater speed of execution of the programs as they are compiled instead of being interpreted. The user can use this component by supplying a program that performs some processing, such as filtering or analyzing packet's headers' contents, and once a packet arrives the virtual machine containing this program is ran on this incoming packet.

The programs are loaded to the kernel via special `bpf` syscalls that are usually wrapped in userspace libraries like `bpftool` or `iptables2` that abstract the exact details of loading the programs and allow for loading and modifying them from the Command Line Interface (CLI). The eBPF is also accessed from the kernel side, especially in network drivers that run the attached programs once a packet is received or transmitted and the programs are loaded with filters determining the packet's course.

Writing an eBPF program is usually done in the C programming language

albeit strongly constrained. Such a program cannot allocate any dynamic memory, must check if the pointer accesses aren't beyond this packet's scope, the recursion is limited to tail calls and there can be no loops (unless unrolled). Tools to make writing these programs easier exist, an example of such are `BCC` [18] and `bpftrace` [19].

Alongside the eBPF virtual machine, there is the XDP in the Linux kernel. This subsystem is what calls the virtual machine and therefore dispatches the packet to wherever it should go. XDP requires the kernel to be at least 4.12 and operates at the second lowest level of the networking stack (the link layer). Only after XDP decides the packet's fate and it is decided to be served, the `skb` is allocated and the packet is passed further into the networking stack.

In context of this project, eBPF is used for processing the XDP program loaded from the userspace to signal the Ethernet driver to offload the packet capturing to the FPGA. Since eBPF is powerful enough to perform advanced filtering on its own, enhancing it with FPGA offload functionalities could be highly beneficial.

1.2.3 SoC platform overview

The board we have used throughout the project is Altera's (now a part of Intel) De0-Nano SoC. This board boasts having both an FPGA and a Hard Processor System containing application grade 2-core ARM processor and various peripherals integrated on the HPS' side. Since there is not much choice on the market when it comes to an FPGA we decided utilize the hardware we already had, hence this board.

Featuring 40 000 logic elements, 5 phase locked loops, a dual-core ARM A9 processor, 1GB of DDR3 SDRAM, and some peripherals including Ethernet and Analog to Digital Converter (ADC), the board is a good starting point for any hobbyist or academic that does not require industry-grade expensive FPGAs. These capabilities are visible in figure 1.3.

Having been released in 2015, the board does not have prominent support anymore as Intel has been focusing more on their newer families of System on Chips (SoCs) (Cyclone 10, Arria, Max). Therefore, our experience with the board has been less-than-optimal due to outdated documentation, packages, and scarcity of users whom we could share our experiences with.

The board has been already used in my Bachelor's thesis [20] and proved to be a good starting point for a hobbyist or a developer that needs a Proof of Concept solution. Due to its affordable price and our previous experience with it, we decided to adhere to it and this time explore networking capabilities and

delve deep into the kernel.

In the future, choosing a board that is fully open-source would be desirable, since one can avoid the vendor lock-in and what that entices - trouble with scarce or diminishing support over the years.

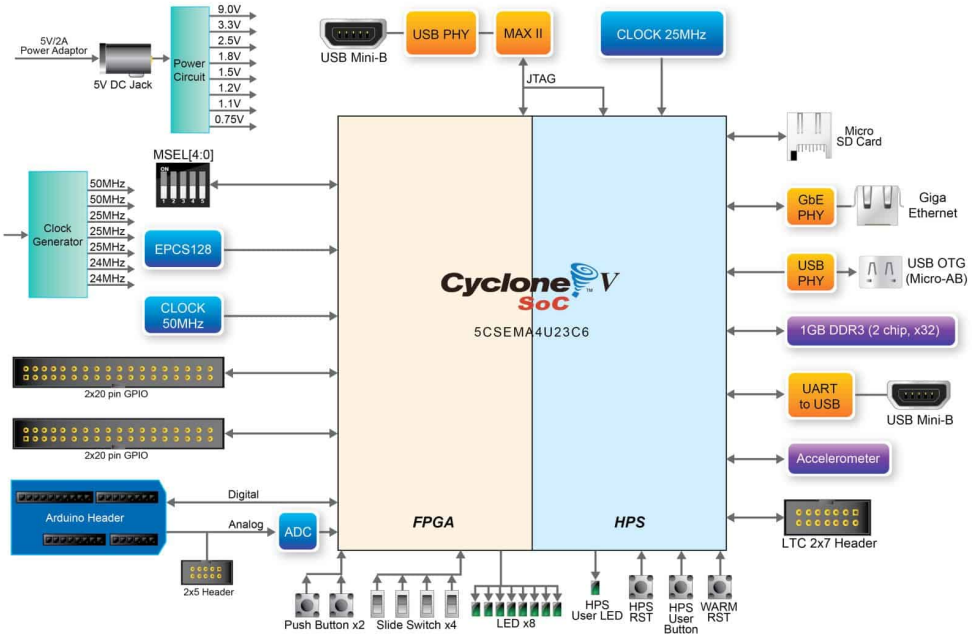


Figure 1.3: Overview of the target platform.

1.3 Problem

As discussed at the beginning of this chapter and in section 1.2, with increase of Edge Computing devices and the ubiquitous presence of Internet thanks to IoT, also rises the need for fast and power-efficient packet acceleration. This can obviously be done with help of ASICs but as discussed earlier, they have their shortcomings and the development cycle is too long or the cost is too high when one deems it necessary to add more features.

Therefore, one may choose a slightly more expensive alternative - FPGA and offload part or whole of packet processing to it.

1.3.1 Original problem and definition

- How can we make best use of FPGA to accelerate network ingress packets, compared to the traditional way in Linux?
- Is using an external accelerator feasible in terms of resource utilization and power consumption?
- How does varying the clock, burst and packet sizes influence the performance and power consumption of such system?

1.3.2 Scientific and engineering issues

Ideally, one would not need to augment the kernel and just create a kernel driver and route all the traffic via the FPGA from NIC. However, we need to add some modifications to our NIC's kernel driver to ensure the packets are not put through the userspace and the rest of kernel network processing chain, but instead served by our FPGA. Moreover, the end user would probably like to have an extensible system independent of the actual hardware and FPGA model.

1.3.3 Scientific contribution

The thesis aims to deliver an extensible packet capturing and filtering acceleration system for FPGA-equipped embedded devices. Apart from providing an integrated system solution, it compares how differing burst sizes, bus widths and clock frequencies can improve or worsen the performance of our system. It also measures how different parameters can influence the power efficiency of the solution. Because the Register Transfer Language (RTL) code is platform-agnostic, it can be ported to a different platform, vendor or even work with a different operating system as long as its registers will be properly interfaced. It may also be a good study of Avalon-MM bursting capabilities and unveil that the implementation is non-trivial even though it may look so at a first glance.

1.4 Purpose

The purpose of this project is to probe how one may use FPGAs for packet acceleration and create custom Linux solutions to reduce total computing power and allow for easy extensibility. The end user can be a hobbyist, a

company that would like to use Edge devices to reduce the total workload on Cloud centres, or perform all the computations at the Edge instead.

Also, this project aims to reduce the total power consumption by reducing the processing workload on the CPU and offloading it to the FPGA, in effect reducing the environmental impact such computations have. If it proves to be unable to reduce the total power consumption, the project will provide guidelines and data that can be used to choose proper parameters when creating a future alternative solution.

1.5 Goals

The goal of this project is to deliver an extensible Linux packet acceleration and capturing solution that uses the on-board FPGA. Apart from providing insights into the process of network packet capturing, it provides experimental results that come from modifying different parameters, such as clock speed, burst size and packet payload size.

1. Develop an extensible and portable open sourced RTL code.
2. Provide necessary stmmac Ethernet driver patches and distribute them.
3. Develop a `.pcap` capturing program.
4. Develop a FPGA packet capturing testing program.
5. Provide a study of varying the burst size, packet size and the clock speed on the packet capturing speed, contrast it with the regular in-kernel path and measure current consumption in each case.

1.6 Research methodology

As the problem of the research already hinted, apart from an implementation of an FPGA accelerated capturing solution, we measured how effective modifying different parameters is - a quantitative study. Also, since we were based on the works of previous people to assess how our approach would fare compared to theirs (the baseline) we introduced an analytical element. Finally, this explores a previously unrealized approach to the problem of packet acceleration so we think it could be categorized as an exploratory study as well.

1.7 Delimitations

The thesis does not aim to prove the superiority of using an external accelerator for packet capturing and instead provides insight into how different parameters can influence the capturing conditions. Only a single platform (De0-Nano SoC) was tested and the kernel was modified specifically for this board and the Micrel `stmmac` Ethernet kernel driver. Features related to filtering the packets and their classification are absent due to the limited scope of this thesis and the considerable effort it would take to implement them optimally. Doing so could be a part of a commercial solution even though could be started as a proof of concept research project. Lastly, eBPF was used only as a hook to attach packet capturing solution even though it could do some preliminary filtering of its own and drop some packets to even further reduce the load on the FPGA. This is beyond the scope of this thesis since it would require adding considerable subsystem and assessing how it interacts with the remainder of the system.

1.8 Ethics and sustainability

Since all research should be ethical and should strive to improve our quality of life and well-being, this research tries to fulfill these goals accordingly. Even though the topic of hardware packet accelerators might not strike as one that can raise concerns regarding ethics, it covers the cornerstone of all data transfer in nowadays world - network packets. Despite users' data being encrypted and encoded, packet sniffing devices or taps in the network are a vulnerability and should be mitigated [21].

Packet sniffing is ethical if done by the network administrator and serves the improvement of the network endpoint or a particular device. Hence, we believe that the ethical considerations regarding this project can be placated, as the device is not efficient enough to serve as a powerful sniffer - it is rather used as a proof-of-concept low-power capturing system.

With the rapid increase in the number of embedded devices and the growth of the Internet of Things, the need for sustainable development and upkeep of electronics is also soaring. Crucial is not only manufacturing the devices in a conscious and aware way but also designing them with a long lifespan and with power efficiency in mind. Additionally, efficient usage of available resources, such as FPGAs, ASICs, GPUs, and cryptographic chips is on par with their environmentally aware design.

Taking the above into account, the design and realization of packet capturing solution in an embedded system should be as concerned with the performance as with the power efficiency. The main issue with such accelerators is that they require additional power to function and that might be sub-optimal if one wants to reduce power at all costs. There are, however, cases where this cost is mitigated and the FPGA outperforms the CPU [22, 23]. Therefore, one should carefully assess whether using an accelerator is necessary and beneficial for the task at hand. CPUs can usually realize the same tasks albeit much slower, so this trade-off is to be considered.

The thesis addresses these concerns and measures the impact that this offload has on the power consumed compared to the regular packet capturing and processing in the CPU.

Lastly, the research does not present any personal data and operates in a synthetic environment. The results of this work should not, however, be used for ethically opaque activities such as packet snooping.

Chapter 2

Background

In this chapter, we will focus on the topic of network packet acceleration, eBPF, and overall Linux kernel networking. We will also cover IoT OSes and how they came to be. Some background on various acceleration platforms is presented, especially in the context of networking. Finally, packet capturing format `.pcap` is introduced and discussed and the Avalon MM protocol used for data transferring in the PL code is presented. Even though most of the groundwork was showcased in the preceding chapter, there remain some nuances that require our attention. We will also discuss the relevant background study in this area so that we can compare how this works fares against them.

2.1 Networking on embedded devices

In past, embedded devices were usually not necessarily connected to any network, be it via Ethernet or using some wireless protocols. More often they would make use of low-frequency radio or very high but short-ranged, such as infrared communication - technologies that better fit these days. If a device required connectivity, it would often be a mission-critical e.g. an Automatic Teller Machine (ATM) which is powerful enough to implement the TCP/IP networking stack and use the Hypertext Transfer Protocol (HTTP) application layer protocol. This changed with the proliferation of cheap Microcontroller Units (MCUs)s and the rise of computing power as well as the invention of more lightweight and robust mobile networking protocols, such as Message Queue Telemetry Transport (MQTT), Advanced Message Queuing Protocol (AMQP) or Constrained Application Protocol (CoAP).

Power is one of the most significant issues when it comes to embedded

devices [24] and it is no different with those that have networking capabilities. As embedded devices often have to operate in low-power conditions relying on a presence of built-in battery supplying them with power, they should be designed to endure possibly the longest on a single charge. Therefore, as shown in the rest of this section, choosing the proper Operating System may be as crucial as the choice of development platform [25] when aiming to improve single-charge uptime.

2.1.1 IoT OSes

As early as 1997, a group of researchers from Olivetti and Oracle Research Laboratory published a paper concerning *piconet* - an ad-hoc network for embedded devices [26]. In their work, they paved the way for novel ways of connecting such small devices taking into account their low processing power, low-rate, and low-range capabilities. Another example of early research that later created a foundation for more mature and advanced operating systems is TinyOS - a system for low-power wireless devices [27]. Being a tempting choice for a Wireless Sensor Network (WSN) or any other distributed system it gained quite a popularity at the beginning of this millennium. The OS is based on the premise of non-blocking calls using a common call-stack for handling *events* - an abstraction available in its custom programming language nesC. Culler et al. [28] have shown in their work how to efficiently make use of the non-blocking paradigm of this system to allow for sophisticated networking capabilities in dot devices based on those developed in DARPA's and Berkeley's Smart Dust project [29].

With the rise of Free Open Source Software (FOSS) we now have a proliferation of different OSes to choose from [30]. From Contiki, [31] that is designed specifically for IoT and ultra low-resource embedded devices, through FreeRTOS [32] that is used for all kinds of devices including mission-critical scenarios, to RIOT [33] and Zephyr [34]. These last two OSes are the most recent and their popularity is soaring due to vibrant and helpful communities, compelling architectures, and programming languages that they support (Rust, C++). One should note that although FreeRTOS does not have built-in networking capabilities it can be easily extended using its ecosystem of libraries.

2.1.2 Linux

When discussing networking on constrained devices, one should not forget about probably the most popular operating system existing - Linux. Even though it was written with mainframes and regular powerful PCs in mind, it is more than capable to be deployed on embedded devices [35, 36, 37]. The real-time patch set that should be arriving in the mainline soon enough is yet another reason why Linux should be considered the first embedded operating system when designing a complex system. The one major disadvantage of this system is that it usually requires much more hardware resources (e.g. RAM, storage) than the alternative systems listed above, including Real-Time Operating Systems (RTOSs) due to resources used by a scheduler, memory-management and protection subsystems, and a myriad of other (configurable) systems.

Thankfully, if one requires real-time capabilities and can afford the overhead ensuing from the usage of Linux, there exist specially crafted patches that change it into an RTOS [38]. Duca et al. proposed hard real-time networking patches that integrated the RTnet networking stack deep into Linux to provide a superb networking performance [39].

The embedded Linux community is a vibrant one, contributing steadily to the mainline kernel and even more so thanks to the adoption of ARM-based PCs among its developers. There is no *one* distribution of embedded Linux, but rather a collection of tools that allow customization and development of a special kernel and root filesystem fulfilling one's needs. The two most popular tools used for this are Buildroot [40] and Yocto [41] with the latter gaining significant popularity and industry support. Both of these tools allow for including different userspace libraries and applications that comprise the final product.

As discussed above, networking is usually based on the basic Linux network stack, but since it is an open-source system it can be replaced or tweaked according to the developer's needs. Additionally, the userspace side of networking applications is usually provided by the aforementioned two build systems and can also be cross-compiled from the source. Thanks to the sophistication of these tools, one can create a truly unique solution and optimize it to their heart's content.

Even though the board used throughout the project utilizes a standard TCP/IP stack, there already exist boards that are much cheaper and smaller, contain an FPGA, and could make use of efficient networking acceleration. Arduino company recently created a board that has an MCU, FPGA, and a

special WiFi chip - Arduino MKR Vidor 4000 [42]. As one can see, the market need for embedded devices that are a mix of software/hardware accelerators is rising, therefore knowledge about these topics may prove indispensable shortly.

Moreover, one could deploy a lightweight implementation of the TCP/IP protocols [43] or even use protocols that are used for communication of the IoT devices instead of the traditional HTTP or Secure Hypertext Transfer Protocol (HTTPS). The protocol stack used depends on the application choice and the resource capabilities of the platform and thanks to such a wide variety, the prospective developer can have sufficient flexibility in this regard.

2.2 Network packet acceleration

With the rise of IoT and the development of next-generation mobile networks, having a wired or wireless networking chip on one's board is nearly a standard. It can be either a specialized auxiliary processor tasked with networking entirely or a CPU, GPU or an FPGA purposed for this task. Having these devices about, one should think about all the additional power that is necessary to power them. That is why extensive research in this domain is still needed, especially when humanity is facing the greatest climatic crisis ever. Why waste computing resources for repetitive computations using a CPU when one can make use of specialized hardware that would perform the same task consuming far less power? One should however remember that running the accelerator itself often amounts to greater power utilization in the system while reducing the usage of the CPU, hence the task in the need of accelerating or offloading should be chosen appropriately.

Although packet acceleration is mostly relevant to big data server clusters and specialized networking rigs that provide the backbone of what we call the Internet, solutions for regular customers also exist [11]. Usually, either entirety or a part of the networking stack is offloaded to the hardware accelerator depending on the requirements. The two most popular hardware devices chosen usually for this task are GPUs and FPGAs and we will be mostly focusing on the latter, however, it is also worth showing the benefits that the former has to offer.

2.2.1 CPU

Even though they cannot be called offloading, there exist packet acceleration solutions that operate solely in the CPU. The oldest one of them is Click - a

modular software architecture for creating routers that allowed for flexibility in creating packet processing paths instead of previously hard-coded processing routes [44]. Another widely used and currently most popular framework is Data Plane Development Kit (DPDK) [45]. It operates all devices in poll-mode and shifts the entirety of packet processing to userspace to remove the overhead associated with kernel-to-userspace transfers. Additional performance gains are due to the usage of huge-pages, cache alignment, core pinning, and disabling interrupts. DPDK can help achieve performance up to 40 Gb/s, as shown by Zhang et al. in their work concerning the development of 5G User Plane Function [46].

2.2.2 GPU

Since the conception of a specialized Graphics Processing Unit to accompany the CPU, offloading the computations on everyday machines became common. For quite a time they were used mostly for rendering 2D and 3D graphics. However, with the rise of interest in domains of Machine Learning (ML) and Deep Learning (DL) popularity of using GPUs as regular processing units soared [47]. Such computations are called General Purpose Graphics Processing Unit (GPGPU) and they can be performed in several programming languages, the most popular of which is Compute Unified Device Architecture (CUDA) [48]. Similarly, frameworks such as PyTorch or Tensorflow rely heavily on GPU's parallel capabilities and leverage it to perform heavy operations like matrix multiplications [49, 50]. DL has been such a resource-demanding domain that Google devised its acceleration platform only for this problem - a Tensor Processing Unit (TPU) [51].

The availability of GPUs and their relatively low cost made them the perfect choice for creating clusters of computing units ready to be programmed in a language of choice, for instance, OpenCL [52]. A major advantage that GPUs have is that programming them is quite similar to programming a regular CPU with the distinction that special care needs to be given to divide the resources between small computing units. However, the major disadvantage that is highlighted by some works in this chapter is the overhead of CPU to GPU transfers and cache coherency [53]. In past, the author has stumbled upon this limitation when developing a GPGPU library for a BeagleBone Black GPU [54].

In the context of network acceleration, GPUs have been used mostly for offloading the most resource-intensive parts of packet processing. For instance, the operation of routing the packets was entirely accelerated by

PacketShader [55], a library that utilizes CUDA for computing batches of packets at a time in the GPU. The throughput performance is assessed to be around 40 Gb/s which is a big improvement over a regular CPU 10 Gb Ethernet. Similarly, in another accelerator project - APUNet, the computations are performed in batches and the IO is done by CPU [53]. The data is transferred between CPU and GPU using shared memory which is managed by DPDK. The performance in this scenario is usually lower than that of the CPU alone which shows that acceleration may not always be desired and sometimes may even be detrimental to performance.

2.2.3 FPGA

Similarly to GPUs, FPGAs have first been used for operating as standalone processing units, be it in a Radio or specialized avionics chip [56]. When used as an accelerator, they suffer from similar bottlenecks to GPUs as the data has to be usually gathered from or transmitted somewhere else. The overhead ensuing from these transfers varies heavily between platforms but can be minimized with good programming practices, proper choice of interfaces, bus widths and appropriate clock speeds [57, 58].

FPGAs find wide application in commercial products, such as Intel's FlexRAN 5G/LTE network stack, especially the lower parts of the stack responsible for precoding and Forward Error Correction (FEC) [59]. They are also prevalent in SmartNICs that are present in big tech companies' Cloud datacenters [60]. Additionally, existing frameworks such as DPDK support offloading packet processing to FPGAs, therefore gains from acceleration can accumulate. Another heavily researched topic is using FPGAs for packet classification [61] and assuring proper Quality of Service (QoS) in the network [62]. In these solutions, FPGA is also usually only a part of a grander processing system.

Apart from FPGAs there exist regular network accelerators that are in principle ASICs designed to perform one task for the entirety of their lifespan. The obvious downside of these accelerators is that they cannot be modified and will be performing the same task until they are scrapped.

2.2.4 NoC

As important as it is to choose an acceleration platform and which parts of the stack to accelerate, it is also crucial to implement the acceleration engine in a coherent and well-communicated way. Packet switching and routing

are also topics that play a fundamental role in designing the acceleration engine and crafting the Network on Chip (NoC). As shown by Abdelfattah et al. [63] choosing a proper architecture for packet routing in the FPGA can achieve 5 times higher processing bandwidth and 3 times lower silicone utilization. The processing speeds of such solutions for an Ethernet switch or a packet processor reach 400 Gb/s and 800 Gb/s respectively [63, 64] which is a considerable speed considering that the baseline processing speed of the regular Linux kernel can rarely reach 10 Gb/s [65].

2.3 Constrained devices and power

In past, there has been scarce research regarding packet acceleration on constrained devices. Even though FPGAs are widely used as cryptographic engines for computing RSA, SHA, and other key parts of the system, they usually don't encroach onto other segments of the networking stack [66]. Thankfully, with the rise in the popularity of IoT and Industrial IoT, research in this domain has become gradually more desirable.

Power is consumed by the devices usually depending on how much they are utilized, for example by switching off external accelerators or utilizing sleep states. Disabling unnecessary peripherals is probably the easiest to achieve and provides greatest gains in terms of consumed power, in turn sacrificing precious cycles it takes the device to be back online. Sleep and low-power states are another approach to this problem and they are less invasive than a total power-off [67]. They are usually implemented in every microcontroller device family and they usually have at least 3 states (a normal power utilization state, a sleep state and a deep sleep state). One can also save power by constructing clever algorithms and data sizes that are best suited for given processor architecture and model [68].

The earliest works that accelerate networking functionality in such systems do so by offloading the networking part onto an RTOS. Maruyama et al. build upon previous research in the domain of FPGA-based RTOSes that are accelerated only partly or in entirety [69]. In their work - Advanced Real Time Embedded Silicon System Operator (ARTESSO), they offload a part of the TCP/IP stack responsible for memory copy, TCP checksum, and header rearrangement and leave the protocol processing to the software. Although they do not compare performance, they conclude that they achieved over 7 times energy reduction over existing commercial firmware RTOS.

Another project that utilizes an FPGA for the realization of a part of the networking stack does so by providing a Network Packet Filter (NPF)

implementation [70]. They use an SoC from Microsemi comprising an ARM Cortex-M3 hard processor and an FPGA that is connected to a Institute of Electrical and Electronics Engineers (IEEE) 802.15.4 transceiver via Serial Peripheral Interface (SPI). Even though they obtained only less than 20% reduction in processing time, it still proves that prudent choice of acceleration scheme can be beneficial.

Gomes et al. in yet another of their works presented CHAMELIOT - a platform for interfacing IoT systems with FPGAs [71]. They base this system on a popular processor architecture - RISCV and provide an environment abstraction layer that allows for easy Application Programming Interface (API) calling from the upper layers. This way they addressed concerns that arose from the industry's skepticism about the adoption of said systems. With over a twice-fold increase in performance, they proved that creating an extensible solution that can offload any computation to hardware is feasible. Their framework is capable of working with OSes such as RIOT, Zephyr, or FreeRTOS. Even though they did not probe any particular networking functionality, solutions such as mine could be deployed alongside theirs.

Since we are in the domain of embedded systems, it would be prudent to mention the power and resource constraints that these ensuing devices impose. A proper choice of architecture and algorithms can help tremendously in meeting these requirements. Czapski et al. have proven that a proper choice of parallelism can help reduce both the static and dynamic power of the FPGAs [72]. Similarly, a choice of scheduling algorithms in the embedded systems may have a significant impact on the total power consumption [73].

Even though there are papers that go into the possible gains due to acceleration [74], there was no research that would go into details of the power consumption of embedded devices that perform network offloading. This might be due to a lack of adequate interest in this domain yet or because when one decides to use an additional processing chip, they are not strongly concerned about the increased power requirements. Therefore, the research proposed here might provide a beneficial contribution in this area.

2.4 *.pcap* file format

To have a unified way of storing networking data for offline analysis and comparison, the `.pcap` format [75] was conceived by the developers of the *tcpdump* program and the *libpcap* library [76]. This is now a de-facto standard capturing format and all popular network analysis tools, such as Wireshark support it. It follows a simple scheme in which the entire capture is prepended

with a header and every packet is prepended with a special header. The file ends with the last packet and no additional payloads are present at the end.

2.4.1 File header

The file header contains all necessary information for interpretation of the data contained in the packet, notably the link type, format version, and a bitmask indicating if a Frame Cyclic Sequence (FCS) is appended to each packet.

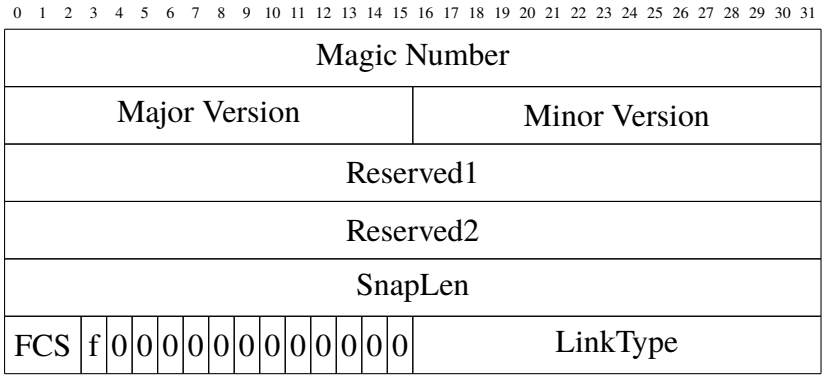


Figure 2.1: .pcap file header format

2.4.2 Packet header

Similarly, the packet header is quite simple and contains just a couple of fields: the current time in seconds and microseconds that elapsed since the Unix epoch. It also contains the original packet length and the captured packet length and these two might differ due to truncation during packet capture.

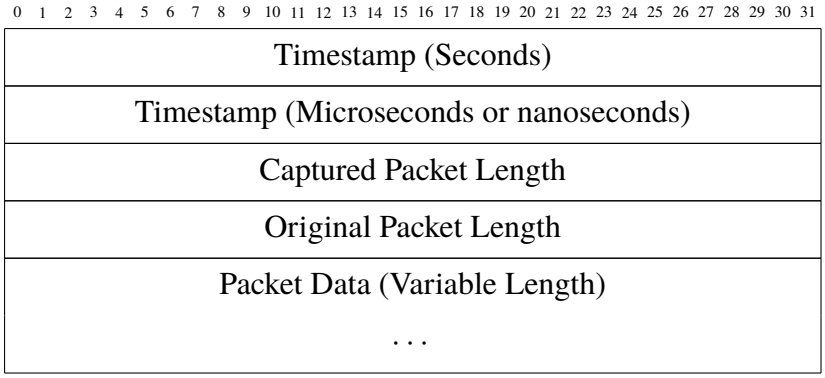


Figure 2.2: .pcap packet header format

2.5 Avalon MM protocol

To fully grasp the solution and the various implementation decisions, one should understand the Avalon Memory Mapped protocol. It is used mostly in Intel's IPs alongside Advanced eXtensible Interface (AXI), Advanced Microcontroller Bus Architecture (AMBA), and Wishbone protocols and is used for communication with a memory-mapped peripheral, such as a set of registers on an FPGA. For streaming capabilities, there is a special variation of this protocol called Avalon ST.

Bursts are sequences of rapidly transferred data using a single interface `write` command. The protocol is fairly simple if no bursts are present, but grows in complexity when they are added due to various corner cases that the implementer must cover. To be the most optimal, they can be of differing sizes, depending on the bus capabilities, and can also differ in width. These characteristics will be probed and analyzed in the future chapter(5) where we discuss the results.

Both transaction sequences are best illustrated by figures 2.3 and 2.4. In these transactions, we act as a host and the outside world is our agent.

2.5.1 Read sequence

The read sequence commences with filling the `address` and the `burstcount`, and then submitting the `read` request. The read is held high as long as `waitrequest` signal is asserted from the agent and is pulled low the next cycle `waitrequest` goes down. The `readdatavalid` signal informs the host that the data present on `readdata` is valid and the host can read it. Each cycle this signal is asserted the data is changed until the agent has transmitted `burstcount` data and the transfer ends.

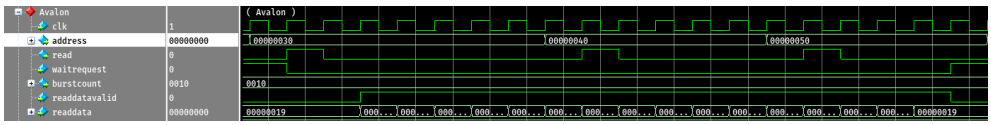


Figure 2.3: Avalon MM Read sequence as visible on the host.

2.5.2 Write sequence

Similarly, the write sequence makes use of the `address` and `burstcount` signals, and when they are prepared, it asserts the `writes` signal. The

agent then knows that the data to transfer is valid and decides whether it can accept it or the host must wait and hold the data constant by asserting `waitrequest`. As soon as `waitrequest` goes down the host knows it can submit the next data on `writedata`. When the `burstcount` of data is transferred, the onus is on the host to submit yet another transaction with `burstcount` of data.

The `burstcount` parameter can be either in words or in symbols (bytes) and this has to be agreed upon on both sides of the communication, usually set in the Platform Designer tool.

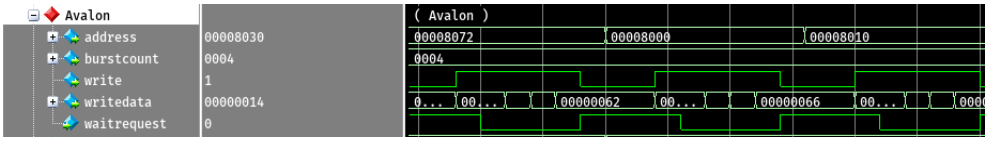


Figure 2.4: Avalon MM Write sequence as visible on the host.

2.6 Summary

As shown in this chapter, the topic of networking, especially in embedded devices is a complex one, as one should take great care to choose the proper environment for their needs. Additionally, choosing *when* and *when not to* accelerate is also of paramount importance, maybe even on par with the choice of *what* to accelerate. The problem of acceleration is ubiquitous and not only restricted to the domain of constrained devices. A careful choice of acceleration algorithm and platform may be as important as the processing power it provides. Nevertheless, the choice of background topics is mostly related to embedded devices since they provide a much better test bench with scarce resources so the coding must not be sloppy and the algorithms have to be thought-through.

Chapter 3

Method for packet offloading performance evaluation

Once the solution was complete and working as expected, the proper research task could be performed. To be conducted properly, the research process had to be meticulously planned and scrutinized once finished.

3.1 Research process

To evaluate the performance of the packet acceleration with various parameters, a coherent research process had to be devised. Adhering to it helped to fulfill the goals assumed earlier and answer questions formulated beforehand.

The research process comprises several steps visible below.

1. **Consider the baseline**
   Before any changes are made to the environment and the target board, baseline metrics should be collected. They will later allow for the comparison of collected parameter modification data. Since there is scarce research on the topic, the baseline values have to be collected empirically.
2. **Measure the external factors**
   Since the measurement is done on the whole system, there are parts over which we don't have control, therefore we need to compensate for them in our measurements. An example of such an external factor is access time from HPS to FPGA when reading or writing the registers. This access time is constant, independent of other parameters chosen and would cause unnecessary bias to measurements.

3. **Choose the parameter to be modified**
   Choosing proper parameters for adjustment is another vital step that impacts the success of the research process significantly. This choice of them is motivated by the literature on FPGA optimization [57, 58] and empirical study.
4. **Define performance measures**
   The performance metrics have to be chosen per the previously selected baseline metrics. An empirical study involving modification of various parameters was performed earlier to assess what to exactly measure and when.
5. **Prepare the environment**
   The environment is adjusted for the modified parameter, be it a full recompilation or just resetting the capture buffer's contents and cleaning old log files. Some parameters require more significant changes whereas others can be tested in an almost unchanged environment.
6. **Perform the test on the target board**
   The test is performed the same for every choice of parameters, hence there is little room for mistakes. The test requires the cooperation of the target and host platforms to send packets over the Ethernet link.
7. **Collect and process the data**
   After the test is conducted, the data have to be collected and processed on the host PC. The data has to be cleaned, parsed, and stored in a format suitable for later plotting. To prevent a scientific error, the data should also be scrutinized at this step and checked if they are logically sound.
8. **Analyse results**
   Once the data is collected and processed, its analysis may commence. The data can be plotted to help illustrate patterns and help form conclusions later. Special care needs to be taken at this step as while a proper presentation of the data may help tremendously in an understanding of the problem, a bad one may hinder its gist equally.
9. **Draw conclusions**
   The last step is to draw conclusions based on the data and plots obtained from previous steps. Assuming the results proved to be valid and relevant, the superiority of some parameters over others may be presented.

The research process presented above (except for the first step) is repeated for every parameter choice. Since some parameter changes require a full recompilation of the environment, they are performed from the most invasive

to the least. This means that first the environment is adjusted for the most intrusive parameter and then it is tested against other parameters that do not require such preparations. The research process could be automated, especially in the steps related to environment preparation, data collection, and processing if one would be willing to.

3.2 Test environment

The test environment comprises two platforms, the host PC tethered to the target platform via an Ethernet cable. The control over the target is assumed via an SSH connection on the host. A socket power meter is connected to the target platform's power supply cord and it is used for power consumption measurement. The data is collected on the target, copied using `scp` to the host, and processed using a Python program. The SD card containing the target image and the FPGA code is flashed and programmed on the host. The software and research-related code are cross-compiled on the host and copied to the SD card.

3.2.1 Hardware/Software to be used

- **Linux host PC** - Used for development and communication with the target board via SSH connection.
- **De0-Nano SoC** - Target board for testing and data collection.
- **Virone EM-1 Power Meter** - Socket power meter used for current consumption measurements.
- **GCC Linaro and ARM toolchains** - Used for cross-compilation of the bootloader, kernel, and rootfs. Also used for the compilation of necessary other target binaries, such as the `.pcap` tool or FPGA testing code.
- **Python 3.10** - Used as a processing tool for timestamp extraction from trace logs coming from the target platform.
- **Quartus IDE** - Development IDE containing SystemVerilog HDL synthesis tools and FPGA binary image generation. Also provided the SignalTap tool used for in-system real-time debugging.

- **Modelsim simulator** - Used for simulating the RTL code for all synthesized modules.
- **Vim text editor** - The most important tool in the project, used for all text editing in the project.

3.3 Data collection and analysis

3.3.1 Data collection

Timestamp data is collected using the built-in tracing kernel functionality but had to be enabled via `CONFIG_FTRACE` kernel parameter beforehand. `trace_printk` statements containing a label and kernel time in nanoseconds were inserted in crucial processing steps. To compensate for the time to communicate with the FPGA using the Avalon MM interface, we performed several measurements of 100 FPGA accesses and calculated a mean value. The results in the next chapter are compensated by three times this amount (once for starting address of the packet, once for its end, and once for reading the control register). Additionally, the raw cycle count was captured from the FPGA to contrast it with the measurements taken from the kernel. The capturing time is calculated using the equation 3.1, where T is the clock period - a reciprocal of the clock frequency and *clock_cycle_count* is the number of clock cycles it takes the FPGA to process the packet.

$$t = clock\_cycle\_count * T \tag{3.1}$$

For the generation of network traffic the ordinary Linux `ping` command is used with `-s` flag to specify TCP payload size `-i` flag to specify the interval of 0.2s and `-c` to specify a count of 100 packets transmitted. Since the board has a static IP, `dhcpd.service` has to be disabled on the host to prevent from dropping the connection every few seconds. During the test, it was determined that background traffic contained Address Resolution Protocol (ARP) and Dynamic Host Configuration Protocol (DHCP) packets, of which the latter disappeared after disabling this functionality on the host. To not interfere with the test results, these packets are ignored.

To access the buffer containing trace logs, the tracing filesystem has to be mounted using the following command:
`mount -t tracefs tracefs /sys/kernel/tracing`.
Afterwards, the `trace` file contents can be dumped to a log file once the test concluded. After copying the data to the host, it is processed using the Python

script calculating the total FPGA execution time and some other statistical parameters derived from it.

Several metrics are used throughout the project, most important of which is *capturing time* - the total time it takes from the arrival of the packet on the NIC to being in a memory buffer available for usage from the userspace. *Processing speed* is sometimes used interchangeably to describe the same quantity. *Time-per-byte* is an additional metric that describes the total time spent in the FPGA divided by the number of bytes processed for this packet.

Since the mains socket power measurement tool does not provide sufficient accuracy of the power reading, we use it for measuring current and then calculate power using the equation 3.2. The resistance was measured under a typical load using the same meter and then assumed to be constant when calculating dissipated power. The current was measured during each test and if it fluctuated between two values, the highest of them was chosen.

$$P = I^2 * R \tag{3.2}$$

3.3.2 Data analysis

After the data collection and processing, it can be analyzed. Plots and charts are important tools for easing the analysis and conclusion-drawing process as hidden relations between data may be uncovered. A standard statistical set of tools for data analysis is used to provide a mean, standard deviation, median, and minimums and maximums from every test result. To better illustrate the performance implications modifications of test parameters may have, an additional metric was developed - time per byte. The equation 3.3 illustrates how it is calculated.

$$t_{per\_byte} = \frac{t_{mean}}{packet\_size} \tag{3.3}$$

Chapter 4

Implementation

This chapter describes the solution in more detail, starting from the hardware part where it explains the capturing logic and finishes with an explanation of how the software part was designed. Decisions that shaped the resultant project are also presented and discussed along the way.

4.1 FPGA design

The programmable logic part was designed and tested on the De0-Nano SoC development board. The design of the system is based on the Golden Hardware Reference Design provided by Altera on a CD accompanying older boards or available on their website [77]. This design provides a good starting point for the Cyclone V series and this board in particular. It provides a top module, pin-out and a `soc_system` - an abstraction that allows for near-seamless integration of various Intellectual Property (IP)-components available in the Intel Quartus Integrated Development Environment (IDE).

Intel's Platform Designer is a tool used for designing the SoC system in a high-level manner. We use it for adding support for necessary bridges and integrating our custom IP component - `bpfcap_fpga`. It is GUI based, hence we were able to avoid some common errors that are present when designing complex systems, for instance, omission of signal assignments. The tool generates necessary bindings, and translations between protocols and provides other glue logic to communicate itself with the Hard Processor System (HPS).

For all of the programmable logic that we have written, we used SystemVerilog, which is much more robust and prevents many coding errors that its younger sibling Verilog does not avoid. We also decided not to use

VHDL programming language for this task as it is overly verbose. Most of the logic in the design is clocked, apart from some minor parts where for bit selection or conditional assignments combinatorial logic was more appropriate. This way we prevented the creation of latches and the propagation of unwanted delays in the system.

Although the components comprising the system could already be seen in figure 1.1, they are magnified in figure 4.1 and will be discussed in more detail in this section.

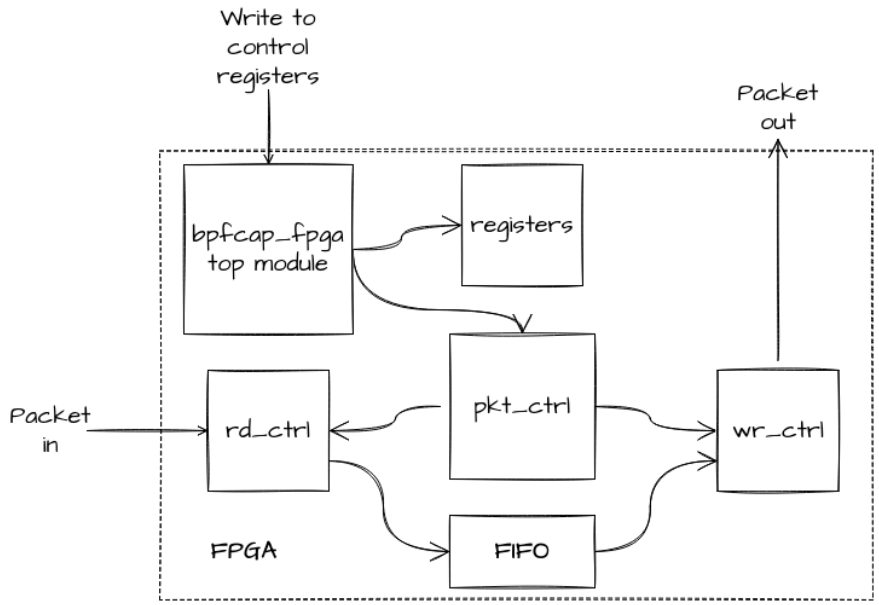


Figure 4.1: Diagram of FPGA modules and logic flow.

4.1.1 Top module, registers and pkt_ctrl

Starting with the top module, one can see that the module communicates with the outside world using the Avalon MM protocol [78], assuming the role of the host twice (once for reading and once for writing control) and once as the agent (for communication as the memory-mapped device from the Linux side). Apart from instantiating necessary components and detecting a start of a new transaction, the acts as the *glue logic*.

There is a standard register module, that holds several registers that control the internal system state and also inform the external components whether it is still busy processing or is ready for receiving more data. The system also contains a state machine that controls the execution of both read and write control modules. Thanks to it, the system knows when to start and stop processing. Additionally, there is a minor module for timestamp generation that contains just two counters, one for seconds and another for nanoseconds.

The user interacts with the top module by providing the address of the register they wish to modify and the data. Once they write to the *pkt_end* register, the Programmable Logic (PL) knows it should start processing.

4.1.2 Read control

The reading module is responsible for obtaining the packets from the kernelspace. Once it starts, the state-machine enters the `READY` state, as visible in figure 4.2, and the loading of new packet data from RAM commences. It knows from whence to read the data as the register module has already latched it beforehand. The data is segmented and read in bursts (a parameter that is modified during testing). When a burst is finished, either a new one is triggered or the transfer finishes - this is controlled by the parameter `total_size` which is set upon starting the Finite State Machine (FSM).

The communication with Avalon MM is done as described in the section related to bursting reads 2.5. Whenever `readdatavalid` signal is high, the data is ready to be read from the interface and `burst_segment_remaining_count` is decremented by the size of the data read in bytes. Conversely, a `waitrequest` prevents the module from loading any more data until it is pulled low.

The data read from the RAM is sent to a First-In First-Out (FIFO) (an Intel's IP) where it waits for receiving by the write control module. The FIFO provides an `almost_full` signal to indicate it will not receive any more data. In such a case, the ingress of data halts, and the logic waits until the write module sufficiently empties the queue.

4.1.3 Write control

Even though originally planned to be equally simple to as reading counterpart, the write control module proved to require more fine-grained control over stalls and packet flows. Once it is triggered, sends a `.pcap` compliant header containing the timestamp and the transmitted packet size. Afterward, the

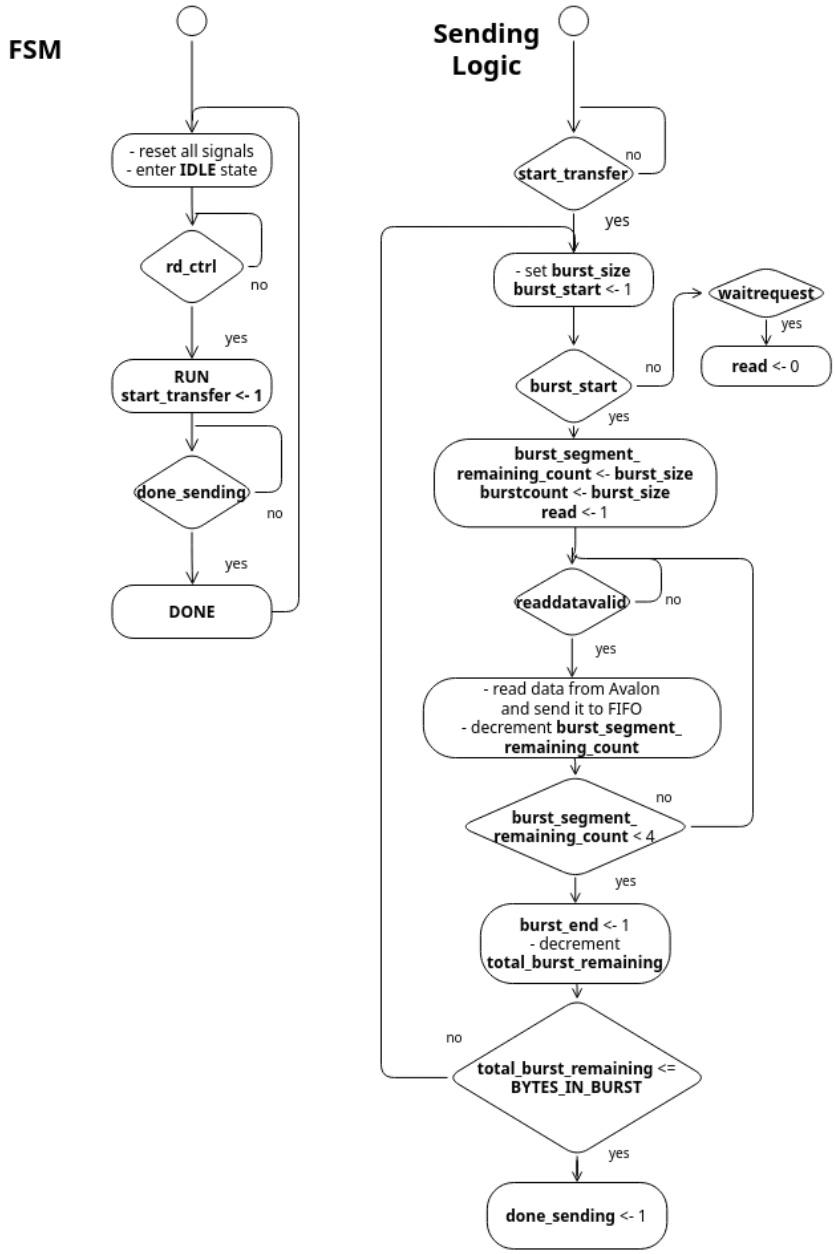


Figure 4.2: Flowchart of Read Control FSM and bursting logic.

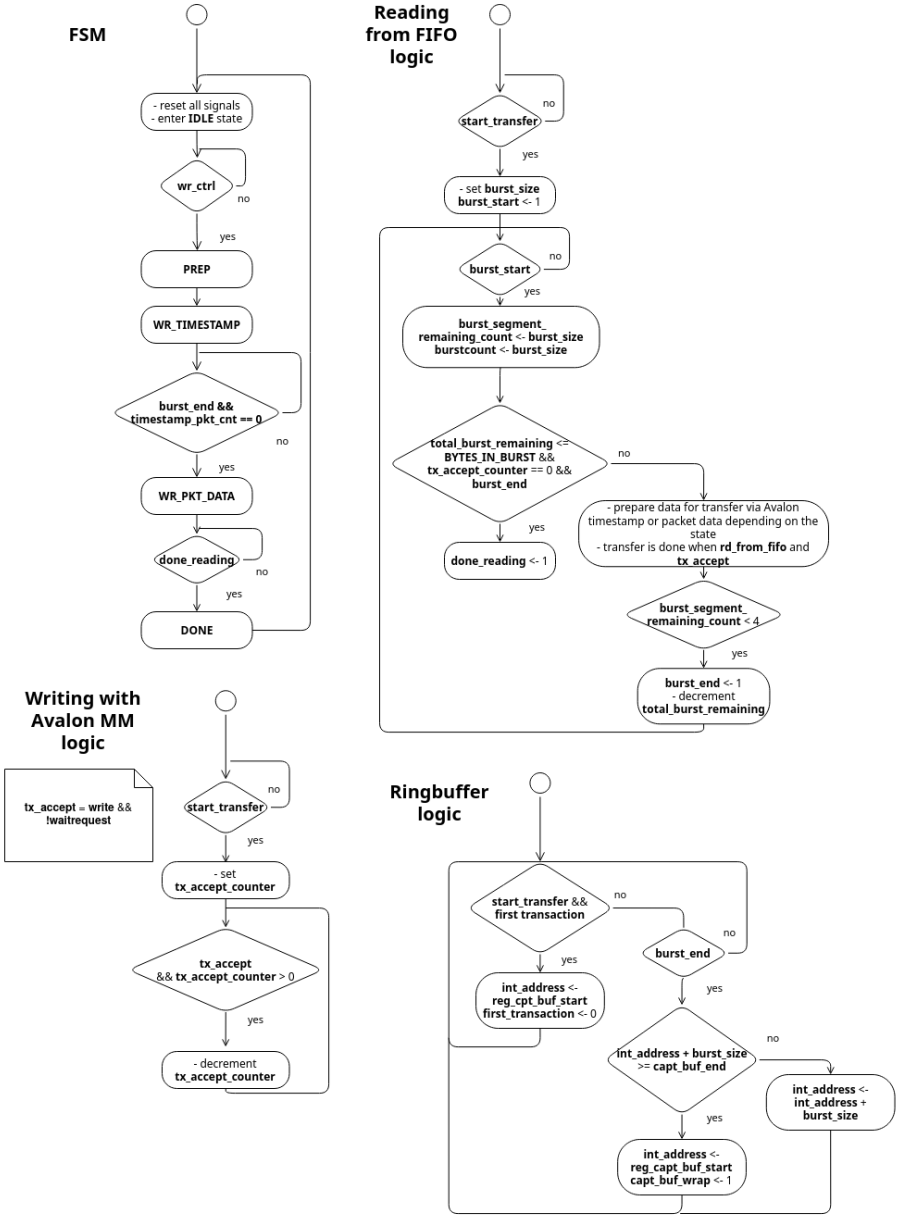


Figure 4.3: Flowchart of Write Control FSM, reading from FIFO, writing to Avalon MM agent and ringbuffer.

regular packet contents follow as long as the FIFO is not empty and the output interface does not signal a stall through a `waitrequest` asserted high.

The transfers are organized into bursts of a preset length controlled by a high-level parameter. This parameter is modified in the succeeding section concerning the results. The problem of stalling of the write pipeline is solved with skid-buffers - encapsulated registers that detect that a stall occurred and hold the value until the flow can proceed [79]. The overhead induced by using these buffers is negligible, hence they prove to be superb for high throughput applications such as this.

This module contains two states `WR_TIMESTAMP` and `WR_PKT_DATA` as different logic is necessary for transmitting the `.pcap` header and regular packet data. The FSM visible in figure 4.3 shows the transition between these states.

Packets are segmented into burst size units and transmitted over the Avalon MM interface, it is performed similarly to its reading counterpart, except for special handling of timestamp transmission and possible stalling on the receiving end. Therefore, the logic is significantly more complex and constitutes a major part of the project. Flowcharts in figure 4.3 illustrate these two paths with somewhat greater granularity. Appendix B contains full code for this module for reference.

The module also contains a ringbuffer functionality to allow for the continuous capture operation. When the write address is near the end of the buffer, special logic determines whether a transaction split should occur and how to perform it. Afterward, it resets the write address to be the beginning of the buffer. Figure 4.3 depicts how it is done.

4.1.4 Simulation

The programmable logic was simulated using two simulators - Intel's Modelsim and SignalTap. Modelsim simulator supports only the synthesizable subset of SystemVerilog but that was sufficient for our needs as we did not need any high-level constructs offered by this language.

The code is organized to allow compilation and running of the simulation from the CLI in mind for possible future integrations in a bigger build system. Controlling running the Modelsim simulation is done by setting an environmental variable.

The testbenches written test the components in isolation as well as test the entirety of the solution. Nevertheless, most of the system-level testing was done on the hardware using the SignalTap logic analyzer due to the shortcomings of the simulation tools. Such shortcomings include a four-state logic whereas on an FPGA there are only two states, another one is a

default width-extension of variables in Modelsim, whereas on the hardware they are truncated. One major drawback of Signaltap is that without a paid subscription for Quartus IDE, each traced signal modification resulted in a full recompilation which is quite time-consuming.

Nevertheless, both of these tools proved to be indispensable when squashing various logic bugs that are elaborated upon in the appendix A.

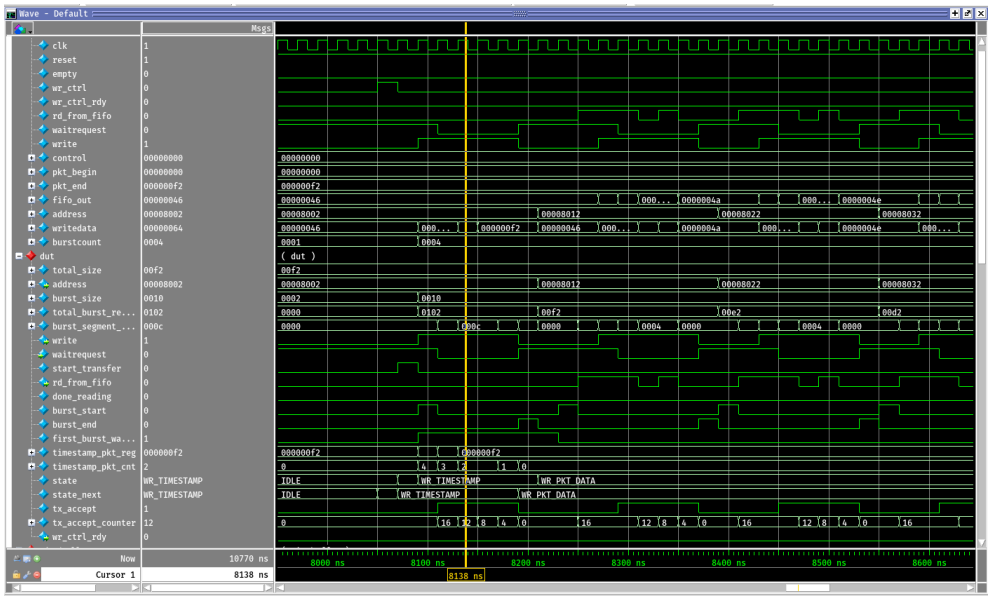


Figure 4.4: wr_ctrl simulation in Modelsim.

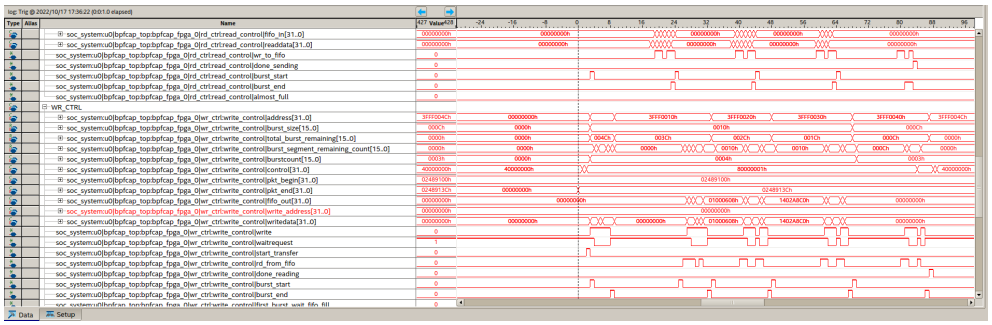


Figure 4.5: SignalTap view of the entire system.

4.2 Software design

4.2.1 System preparation

The software part of the system is composed of the system, driver, and userspace parts. The system part is a uboot bootloader, Linux kernel, and a rootfs. The bootloader and the kernel are forks of their respective origins initially made by Altera and now maintained by Intel [80, 81]. The rootfs is based on Buildroot pseudo-distribution [40] and required the addition of several scripts to allow for SSH Ethernet connections and utilizing *bpfcap* userspace functions.

4.2.2 Kernel driver

To control the FPGA from the networking stack, we had to modify the in-tree Ethernet driver for embedded devices used by De0-Nano SoC - `stmmaceth`. The driver is augmented to initialize the device when the Ethernet device is configured and also when capturing packets that arrive from the NIC. Snippets of code responsible for FPGA configuration and packet capturing are visible in listings 4.1 and 4.2.

First, a memory region for FPGA registers has to be requested from the Kernel, and then `ioremap`'ped. Once it is done, raw FPGA registers can be written to via `writel` functions. Upon reloading or unloading the driver the device is unmapped and the memory is freed.

```
#define BPFCAP_PHYS_ADDR 0xc0010000
#define BPFCAP_BUF_START 0x3fff0000
#define BPFCAP_BUF_SIZE  0x10000
#define BPFCAP_REG_SIZE  0x1C
static void __iomem *bpfcap_fpga_dev;

int stmmac_open(struct net_device *dev)
{
    // ommited

    // reserve FPGA iomem
    struct resource *region = request_mem_region(
   BPFCAP_PHYS_ADDR, BPFCAP_REG_SIZE, "bpfcap_fpga_priv");
    if (!region)
    {
        printk("Failed to allocate memory region");
        goto init_phy_error;
    }
```

```
18      bpfcap_fpga_dev = ioremap(BPFCAP_PHYS_ADDR,
        BPFCAP_REG_SIZE);
19
20      // set the buffer
21      printk("Writing 0x%08x to addr %p", BPFCAP_BUF_START,
        bpfcap_fpga_dev + 0xC);
22      writel(BPFCAP_BUF_START, bpfcap_fpga_dev + 0xC);
23      printk("Writing 0x%08x to addr %p", BPFCAP_BUF_SIZE,
        bpfcap_fpga_dev + 0x10);
24      writel(BPFCAP_BUF_SIZE, bpfcap_fpga_dev + 0x10);
25
26      // ommited
27  }
```

Listing 4.1: Example of FPGA configuration from the Ethernet device driver.

Only after a proper XDP program that returns a custom exit code is loaded, the packets start to flow through the FPGA. This is done in the function `__stmmac_xdp_run_prog` which is run each time a ingress packet arrives. Struct `xdp_buff` contains the packet's start and end address as a pointer to virtual memory. Therefore, a virtual-to-physical translation has to be performed and this is done by the `virt_to_page` kernel function and alignment of the resulting address to a page boundary. It should be noted that the code is not portable if the platform has pages that are not of 4 KB size.

Once the physical address of the packet is obtained it can be written to the FPGA and the processing of the packet continues in the hardware. There exists a possibility that the FPGA is still processing the packets even after the `writel` function returns and for this reason, a check to see if the packet is still processing was implemented. When the packet has been processed, the function returns with `XDP_PASS` to indicate that the result of the XDP filtering is successful and the packet can proceed through the kernel. This can be changed to `XDP_CONSUMED` to indicate that the packet has been processed and can be freed without being served by the kernel.

```
1   static int __stmmac_xdp_run_prog(struct stmmac_priv *priv,
2                   struct bpf_prog *prog,
3                   struct xdp_buff *xdp)
4   {
5       u32 act;
6       int res;
7       u64 t1, t2;
8
9
10      printk("Entered __stmmac_xdp_run_prog");
11      printk("Packet start addr 0x%08x end addr 0x%08x", xdp->
```

```
    data, xdp->data_end);
12   printk("Data length %d", xdp->data_end - xdp->data);
13
14   int i;
15   struct page *page = virt_to_page(xdp->data);
16   void* phys_start = page_to_phys(page) + ((int)xdp->data &
     0x00000fff);
17
18   t1 = ktime_get_ns();
19   act = bpf_prog_run_xdp(prog, xdp);
20   t2 = ktime_get_ns();
21   trace_printk("XDP processing time %llu\n", t2 - t1);
22
23   switch (act) {
24   case XDP_PASS:
25       res = STMMAC_XDP_PASS;
26       break;
27   case XDP_TX:
28       res = stmmac_xdp_xmit_back(priv, xdp);
29       break;
30   case XDP_REDIRECT:
31       if (xdp_do_redirect(priv->dev, xdp, prog) < 0)
32           res = STMMAC_XDP_CONSUMED;
33       else
34           res = STMMAC_XDP_REDIRECT;
35       break;
36   /* XDP_FPGA_CAPTURE */
37   case 5:
38       {
39           t1 = ktime_get_ns();
40           trace_printk("Before FPGA %llu\n", ktime_get_ns()
    );
41           u32 reg = readl(bpfcap_fpga_dev);
42           //printk("Writing 0x%08x to addr %p", phys_start,
     bpfcap_fpga_dev + 0x4);
43           writel(phys_start, bpfcap_fpga_dev + 0x4);
44           //printk("Writing 0x%08x to addr %p", phys_start
    + (xdp->data_end - xdp->data), bpfcap_fpga_dev + 0x8);
45           writel(phys_start + (xdp->data_end - xdp->data),
    bpfcap_fpga_dev + 0x8);
46
47           i = 0;
48           /* unlikely */
49           while (reg & (0x1 << 31) && i < 100) // while
    BUSY
50           {
51               reg = readl(bpfcap_fpga_dev);
```

```
52                 ++i;
53                 printk("STILL PROCESSING i=%d", i);
54             }
55             if (i == 100)
56                 printk("TOO LONG i over 100");
57 
58             t2 = ktime_get_ns();
59             reg = readl(bpfcap_fpga_dev + 0x18);
60             trace_printk("Actual FPGA time 0x%08x\n", reg);
61             trace_printk("FPGA processing time %llu\n", t2 -
   t1);
62 
63             //printk("Returned XDP_FPGA_CAPTURE");
64             //res = STMMAC_XDP_CONSUMED;
65             res = STMMAC_XDP_PASS;
66         }
67         break;
68     default:
69         bpf_warn_invalid_xdp_action(act);
70         fallthrough;
71     case XDP_ABORTED:
72         trace_xdp_exception(priv->dev, prog, act);
73         fallthrough;
74     case XDP_DROP:
75         res = STMMAC_XDP_CONSUMED;
76         break;
77     }
78 
79     printk("Handled xdp bpf with return %d", act);
80     return res;
81 }
```

Listing 4.2: XDP processing function offloading work to the FPGA.

4.2.3 Userspace applications

Finally, the last component of the solution is a simple userspace application capable of dumping the captured memory to a `.pcap` file. The application `mmaps` a part of the memory where the dump buffer resides, opens a file, and writes a special `.pcap` file header to which it then appends the contents of the capture buffer - packet data. The user may choose how much memory to save and the name of the file under which to store captured packets.

The other application is used for testing purposes and was used extensively in the early stages of the project when integration into the Linux kernel was still under development. It allows for testing the FPGA code without intervening in

the Ethernet driver code, hence it may be used for testing additional features. It operates by using the reserved kernel memory buffer as the reading and writing destination simultaneously so it might be sub-optimal and not representative of the real scenario because the memory is shared.

Chapter 5

Results and analysis

In this chapter, we present the results of running the packet capture acceleration with various parameters and with varying clock speeds and discuss them. We start by comparing a quantitative study of different packet lengths and how it impacts the capturing time. Similarly, we check how different burst lengths impact this speed. Finally, we compare how increasing the clock speed of the design affects this metric. The power consumption of the system across several configurations is also presented. Lastly, a qualitative comparison of regular in-kernel Linux packet processing is carried out against our solution. All the tests were carried out successfully, following the test methodology described in the section 3.1.

5.1 Major results

Sections below describe how varying the burst length and packet size performed under each clock frequency. Varying the packet sizes was performed for every other parameter and requires no changes in the test environment. Changing burst lengths and increasing the clock frequency requires a full recompilation of the project and some adjustments to the testing harness. For every measurement, we provide a set of statistical variables: mean, median, standard deviation, minimum, and maximum. These are calculated after the data collection in the processing step. Every measurement consists of capturing 100 packets providing a fair level of diversity of the results.

Table 5.1: Results of the measurements with different parameters.
Note: BL - burst length, W - words

(a) f=50 MHz BL=4 W

	Time [ns]				
Packet size [B]	Mean	σ	Median	Min	Max
32	2209	114	2180	1940	2520
64	3033	110	2980	2940	3320
128	4677	136	4620	4540	5060
256	7993	162	7960	7740	8440
512	14587	279	14540	14160	15440
1024	27706	1409	27690	14520	29320

(b) f=50 MHz BL=8 W

	Time [ns]				
Packet size [B]	Mean	σ	Median	Min	Max
32	1658	101	1640	1620	2040
64	2203	105	2180	2100	2600
128	3321	151	3300	3140	3780
256	5510	207	5500	5220	5920
512	9931	555	9980	5740	11140
1024	18683	1379	18940	6880	20640

(c) f=50 MHz BL=16 W

	Time [ns]				
Packet size [B]	Mean	σ	Median	Min	Max
32	1440	50	1420	1420	1780
64	1666	60	1640	1640	1960
128	2339	100	2290	2280	2680
256	3849	182	3880	2320	3980
512	7081	381	7080	3880	7700
1024	13525	427	13480	12960	14800

(d) f=100 MHz BL=4 W

	Time [ns]				
Packet size [B]	Mean	σ	Median	Min	Max
32	1113	50	1090	1070	1270
64	1521	73	1500	1070	1700
128	2335	111	2320	1470	2550
256	3981	186	3980	2370	4270
512	7282	361	7270	4020	7720
1024	13785	681	13850	7540	14640

(e) f=100 MHz BL=8 W

	Time [ns]				
Packet size [B]	Mean	σ	Median	Min	Max
32	824	46	810	790	1000
64	1102	58	1090	810	1270
128	1653	98	1645	1090	1930
256	2760	162	2770	1790	3130
512	4983	307	4965	3110	5960
1024	9366	613	9465	4960	10370

(f) f=100 MHz BL=16 W

	Time [ns]				
Packet size [B]	Mean	σ	Median	Min	Max
32	718	18	710	710	820
64	830	33	820	710	980
128	1160	53	1150	820	1350
256	1947	97	1940	1150	2170
512	3545	195	3550	1940	3880
1024	6711	376	6720	3510	7350

(g) f=200 MHz BL=4 W

	Time [ns]				
Packet size [B]	Mean	σ	Median	Min	Max
32	556	26	545	535	645
64	754	34	745	545	855
128	1169	58	1165	735	1280
256	1988	95	1990	1160	2155
512	3628	194	3633	1970	3840
1024	6925	355	6925	3620	7355

(h) f=200 MHz BL=8 W

	Time [ns]				
Packet size [B]	Mean	σ	Median	Min	Max
32	411	20	405	395	485
64	549	31	545	395	660
128	832	48	825	625	960
256	1390	90	1385	795	1600
512	2505	155	2503	1390	2765
1024	4687	288	4735	2585	5160

(i) f=200 MHz BL=16 W

	Time [ns]				
Packet size [B]	Mean	σ	Median	Min	Max
32	379	11	375	375	455
64	433	13	430	380	520
128	580	24	575	430	665
256	980	56	970	575	1075
512	1782	53	1775	1710	1960
1024	3370	199	3370	1710	3675

5.1.1 Baseline

Baseline is evaluated by measuring the time it takes a packet from arrival in kernel until it is copied out to userspace. This time was estimated to be around **50 microseconds** independent of other parameters that affected processing speed through the FPGA. Time is measured using `ktime_get_ns()` kernel calls which return the time that elapsed since boot in nanoseconds resolution. The power consumption of the baseline is visible in the table 5.2 at row *Board normal state*.

5.1.2 Packet size variation

Packet sizes are varied starting from 32 bytes of payload that was added on top of the Internet Control Message Protocol (ICMP) packet of 40 bytes up to

1024 bytes in size. For every successive test, the payload size is doubled up to 1024 bytes which are near the boundary of a regular 1500 bytes of Maximum Transmission Unit (MTU).

Capturing times of packets of differing sizes are visible in subtables of the table 5.1 where BL stands for Burst Length and W means words. Figures 5.1, 5.2 and 5.3 show this increase in graphical format for every clock frequency. The time presented in the tables and figures corresponds to packet capturing time as viewed from the FPGA.

As expected, with the increase in the packet length, the total capturing time also increases. This is true regardless of burst length or clock frequency and can be explained by the time it takes for the FPGA to process the additional payload. Whereas for small packet sizes such as 64 or 128 the increase in capturing time does not seem to be doubling, it is more obvious for longer packets. This can be explained by the necessary overhead of running the hardware logic that runs independently of the packet size.

5.1.3 Burst length variation

Burst lengths assumes 3 values: 4, 8, and 16 words. These values were chosen empirically also being a popular choice for bursting interfaces when power and low cycle count is desired.

For every burst length, we can observe quite low standard deviation values which hints that there are no outliers in the data. Similarly, the median is very close to the mean, and in some cases, it is even equal which further hints that processing logic runs in a deterministic fashion and with only slight delays. These delays are probably induced by the interconnect logic or stalls on the FPGA2HPS bridge that is used for FPGA to SDRAM and HPS communication.

Whereas when increasing the clock cycle the total capturing speed is almost exactly halved, for an increase in burst length, the time does not drop as significantly. Nevertheless, a gain of around 33% per each length increase is a desired and expected outcome. The reason why the gain is not as big as the one due to the clock frequency increase may be caused by the external interface implementations and the overhead associated with them. One should remember that due to hardware limitations, we were not able to directly utilize the FPGA2SDRAM bridge, but instead had to rely on the FPGA2HPS bridge and Avalon MM to AXI interconnect.

5.1.4 Clock cycle variation

We were able to synthesize code with 3 clock speeds: 50, 100, and 200 MHz, and modified the two remaining parameters for each of these speeds. For each doubling in clock frequency, the capturing time almost exactly halved. This is visible in figures 5.1, 5.2, 5.3 and is most striking in figure 5.4.

An increase in clock cycle frequency directly influences the speed at which the FPGA processes packets and performs the logic operations. Since SDRAM onboard De0-Nano is already clocked with 200 MHz, using a clock slower than that is an under-utilization and a plausible bottleneck when interfacing with the memory. CPU, being another element of the system under test, is clocked at a much higher frequency, hence an increase in FPGAs frequency provided a visible reduction of total capturing time.

The per-byte capturing time decreases with an increase in the clock frequency and burst length which is also something foreseen. While for smaller clock frequencies, the improvements in the per-byte capturing time were greater, they were much less striking as the packet size increased for a higher frequency. This discrepancy may be explained with compensation of the logic overhead that is almost the same no matter the parameter variation. With more total time spent processing packets and keeping internal logic processing time constant, this overhead spreads more evenly across every byte. The result is visible with a flattening of the curve in figure 5.4.

5.1.5 Power consumption

Board stripped of the FPGA code consumes the same amount of current as when the code is loaded. Unloading the Ethernet driver reduces the current consumption from 41 mA to around 36 mA. The power does not change significantly when the clock speed is increased as can be seen in the table 5.2. Between various tests, the current consumption was oscillating around 41 mA hinting that the FPGA code was not inducing any additional load to the system. The board also does not consume any additional power when

5.2 Reliability analysis

The repeatability of a scientific experiment is of paramount importance when one wishes to expand upon the findings from previous research. Small perturbations in a seemingly unrelated part of the test harness can greatly disturb the outcome, in effect ruining the experiment. Great care has to be

Figure 5.1: Capturing time for a burst length of 4 words.

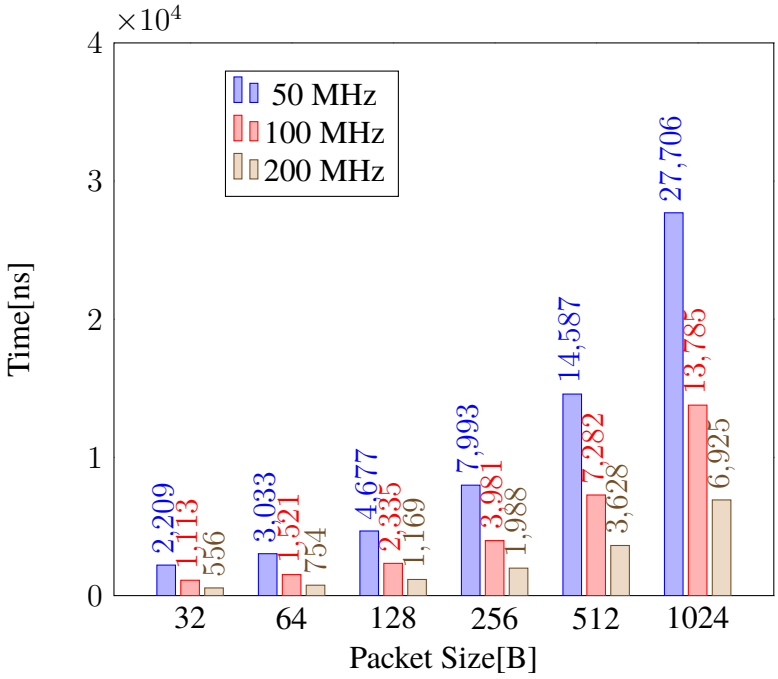


Figure 5.2: Capturing time for a burst length of 8 words.

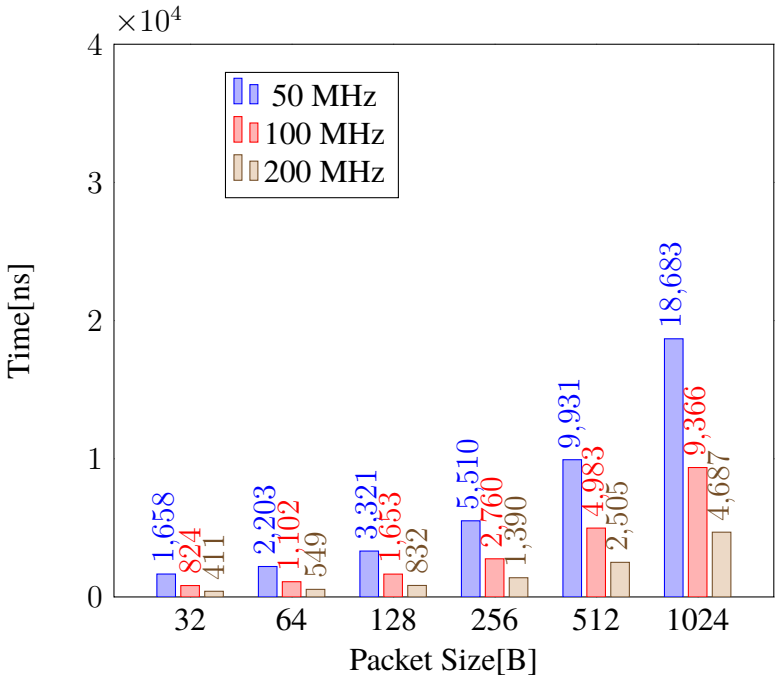


Figure 5.3: Capturing time for a burst length of 16 words.

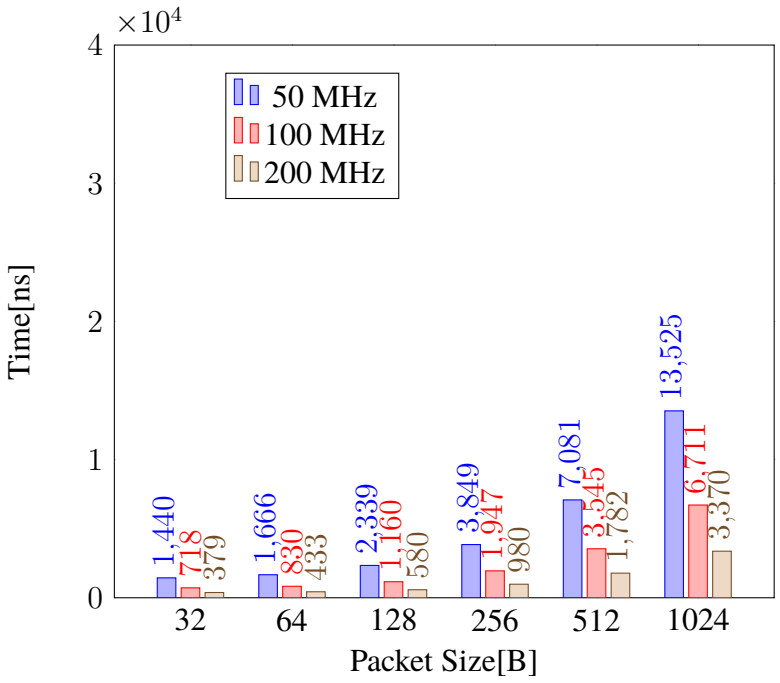
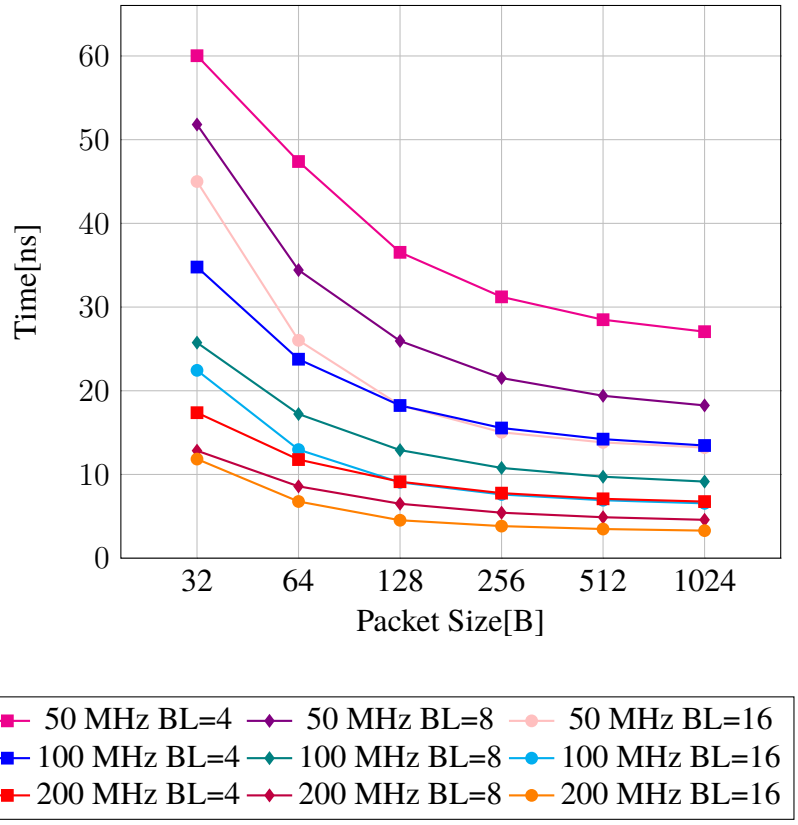


Table 5.2: Power consumption of the target board.

	Current [mA]	Power [W]
Idle: Board normal state	41	4.67
Idle: Board no Ethernet	36	3.60
Processing: Baseline in-kernel processing	41	4.67
Processing: FPGA code at 50/100 MHz	42	4.90
Processing: FPGA code at 200 MHz	41	4.67

Figure 5.4: Capturing time per byte for all parameter variations. Note: BL - burst length.



applied when conducting research such as this, where a human is part of the testing loop.

Performing this research requires changes in only one parameter at once, running the test scenario, and finally obtaining and cleaning the data. Since the system under test is an embedded device, there is not much background noise that could influence the accuracy of measurements and if there is any, the system rejects it. As mentioned in the section describing the research methodology, this noise comes from spurious networking packets on the Ethernet connection that influence only the packet transmission time and not the actual capturing time.

Nevertheless, since this is a regular Linux distribution, some tasks happen periodically and daemons run in the background, possibly interfering with the system. These interferences are, however, not relevant to our measurements as the packets are handled inside IRQs, and kernel tasks have higher priorities than userspace ones.

Measuring across only 100 packets may not be sufficient enough to capture all possible outliers and provide an accurate statistical representation of the packet capturing time. However, repetition of these processes for several configurations at different periods tests the scenario against various system states, effectively reducing the impact of this short measurement period on reliability.

5.3 Validity analysis

The selection of a proper measurement process and metrics is equally important as assuring that the test conditions are repeatable. Therefore, measuring time in the kernel had to take into account the standard inaccuracy induced by the limitations of internal clock resolution. These results are contrasted with much more accurate clock cycle counts performed by the hardware.

Due to the limited resolution of the power consumption meter, this measurement may be highly unreliable and is included in the report only as a cursory finding. In an industrial application, one would use professional equipment for such measurements.

Human error is reduced to a minimum thanks to the usage of data processing automation scripts. If one wished to increase reliability in this aspect, one would need to automate the entire testing environment which makes for a formidable task.

5.4 Discussion

5.4.1 Capturing speed

As presented in the previous section, the packet capturing speed is influenced directly by an increase in the burst length and the clock frequency.

Measurements of the capturing speed from the kernel side show hat the FPGA access overhead is constant and equals about 243 nanoseconds per access. Moreover, since this is asynchronous processing and the CPU does not need to wait for the accelerator to finish processing, it can continue serving the IRQ and processing more work. Even at the lowest clock frequency, the capturing time of FPGA is less than the time between IRQs served in a non-RT Linux kernel which operates at millisecond resolution while the heaviest packets take only up to 3 microseconds to process.

Regular in-kernel packet processing is determined to be around 50 microseconds, measured from the start of the IRQ until the packet's transmission to userspace. Hence, our packet-capturing solution outperforms regular kernel processing by a factor of 16 for the worst-measured case and over 100 times for the best case. It must be noted that kernel adds additional processing on top of packet capturing that our solution is missing and therefore this result is not an equal comparison. Hence, kernel's processing time has a big overhead accompanying its capturing time.

5.4.2 Power consumption

Interestingly, the power consumption does not change drastically with varying the packet size and remains steady even when the clock frequency increases. This is contrary to expectations, since we do not compute anything in the FPGA so it should be not consuming any power, and is a welcome observation given how crucial power efficiency is in the domain of embedded devices. The lack of changes in this area might be due to the small size of processing logic or optimal implementation and not much data being processed simultaneously.

Findings in this area support claim that Possa et al.[57] have made concerning offloading part of the processing to FPGAs to reduce total power consumption in the system. While this project is not offloading a computation-heavy and highly parallelizable algorithm as was done in [23], a small reduction of total capturing speed was expected at lower clock speeds. More research in this area is necessary to formulate a conclusive claim.

5.4.3 Resource utilization

As visible in figure 5.5, the design consumes less than a quarter of logic elements. Most resources are consumed by the write control module and the interconnect between SDRAM and the Avalon MM host is implemented by the reading and write control modules. Details on the exact logic elements used by each module are visible in the figure 5.6.

Flow Status	Successful - Wed Oct 19 23:47:33 2022
Quartus Prime Version	19.1.0 Build 670 09/22/2019 SJ Lite Edition
Revision Name	soc_system
Top-level Entity Name	ghrd
Family	Cyclone V
Device	5CSEMA4U23C6
Timing Models	Final
Logic utilization (in ALMs)	3,685 / 15,880 (23 %)
Total registers	5348
Total pins	230 / 314 (73 %)
Total virtual pins	0
Total block memory bits	541,696 / 2,764,800 (20 %)
Total DSP Blocks	1 / 84 (1 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 5 (0 %)
Total DLLs	1 / 4 (25 %)

Figure 5.5: Total De0-Nano SoC resource utilization.

1	⊟ |ghrd	5233 (1)	4844 (0)	541696	1	230	0
1	⊟ |soc_system:u0|	5025 (0)	4658 (0)	541696	1	0	0
1	⊟ |soc_system_mm_interconnect_1:mm_interconnect_1|	1034 (0)	713 (0)	0	0	0	0
1	|altera_merlin_slave_translator:bpfcap_fpga_0_avalon_rw_slave_translator|	4 (4)	35 (35)	0	0	0	0
2	|altera_avalon_sc_fifo:bpfcap_fpga_0_avalon_rw_slave_agent_rdata_fifo|	38 (38)	66 (66)	0	0	0	0
3	|altera_avalon_sc_fifo:bpfcap_fpga_0_avalon_rw_slave_agent_rsp_fifo|	38 (38)	70 (70)	0	0	0	0
4	⊞ |altera_merlin_slave_agent:bpfcap_fpga_0_avalon_rw_slave_agent|	43 (9)	12 (0)	0	0	0	0
5	⊞ |altera_merlin_burst_adapter:bpfcap_fpga_0_avalon_rw_slave_burst_adapter|	74 (0)	102 (0)	0	0	0	0
6	|altera_merlin_width_adapter:bpfcap_fpga_0_avalon_rw_slave_rsp_width_adapter|	74 (74)	32 (32)	0	0	0	0
7	|altera_merlin_width_adapter:bpfcap_fpga_0_avalon_rw_slave_cmd_width_adapter|	86 (86)	14 (14)	0	0	0	0
2	⊟ |bpfcap_top:bpfcap_fpga_0|	1205 (9)	979 (1)	16384	1	0	0
1	|pkt_ctrl:packet_control|	42 (42)	69 (69)	0	0	0	0
2	⊞ |fifo:fifo_i|	48 (0)	29 (0)	16384	0	0	0
3	|timestamp:ts|	70 (70)	62 (62)	0	0	0	0
4	|register_bank:regs|	112 (112)	192 (192)	0	0	0	0
5	|rd_ctrl:read_control|	147 (147)	183 (183)	0	0	0	0
6	⊟ |wr_ctrl:write_control|	777 (651)	443 (287)	0	1	0	0
1	|skidbuffer:skbf2|	48 (48)	78 (78)	0	0	0	0
2	|skidbuffer:skbf1|	78 (78)	78 (78)	0	0	0	0
3	⊟ |soc_system_mm_interconnect_0:mm_interconnect_0|	1742 (0)	2160 (0)	0	0	0	0
1	|altera_merlin_master_agent:bpfcap_fpga_0_avalon_write_master_1_agent|	9 (9)	1 (1)	0	0	0	0
2	|altera_merlin_traffic_limiter:bpfcap_fpga_0_avalon_read_master_0_limiter|	12 (12)	7 (7)	0	0	0	0
3	|altera_merlin_master_agent:bpfcap_fpga_0_avalon_read_master_0_agent|	29 (29)	1 (1)	0	0	0	0
4	|altera_merlin_master_translator:bpfcap_fpga_0_avalon_read_master_0_translator|	90 (90)	49 (49)	0	0	0	0
5	|altera_merlin_master_translator:bpfcap_fpga_0_avalon_write_master_1_translator|	114 (114)	50 (50)	0	0	0	0

Figure 5.6: Detailed resource usage per module.

Chapter 6

Conclusions and future work

This chapter concludes the work performed in this research. Relevant conclusions are presented, especially those about potential gains from utilizing hardware accelerators in embedded devices. Limitations, being an inseparable part of every research problem have to be discussed and explained. Since the project aims to be easily extensible, some future work is suggested as a definite addition and the priority of these extensions is discussed based on the gained benefits. Finally, reflections on the socioeconomic and environmental aspects of the research are probed in more detail.

6.1 Conclusions

The core goal of the thesis was the development of an extensible FPGA packet accelerator that would utilize the eBPF subsystem. From it, stems its logical continuation of probing the performance of the implemented solution and discussing possible implications of different parameter values. Evaluation of these goals is present in chapters 4 and 5 where justifications design decisions and discussions of the obtained results are provided. Since attaining the primary goal was for a time at a risk, its implementation is a major success and provides a solid groundwork for future developments in this domain. The conception of which parameters to modify took place during its development and therefore proved to be a solid and motivated choice. The main findings include the obvious performance gain from an increase in the clock frequency, regardless of other parameters, up to a maximum reachable frequency of the target platform of 200 MHz. Surprisingly, this increase does not follow in an increased current consumption which may indicate that one could start with a high clock speed when developing a similar solution and not pay the

penalty of wasted energy. However, it must be noted that the system does not use any sleep states which might reduce its total power consumption. Increases in burst lengths prove to also be beneficial to packet processing but not as effective as raw clock speedup. Time per byte is a great indicator of high overhead associated with the processing of small packets and proved that packet sizes of sizes 128 bytes and greater are much more suitable. Lastly, the solution proves to be faster than baseline Linux packet processing significantly. This conclusion is only cursory and to determine systematically the superior implementation would require major changes to the Linux kernel which is out of scope for this thesis.

6.2 Limitations

A few obstacles were encountered during the development of the thesis, some of which are explained in more detail in appendix A. One of these includes the inability to use the FPGA2SDRAM bridge that could not be configured under our software configuration regardless of different Cyclone V register modifications. Therefore, the highest possible bus width of 256 bits between SDRAM and FPGA is not utilized, and instead, only 128 bits are used.

Because the interconnects between Avalon and AXI are automatically generated, we have no control over their implementation and resource usage. They are additionally a closed-source IP of Intel which even further prevents any optimizations to be done in them.

Since there are no low-power states utilized, the solution was not compared to a power-optimized version of regular in-kernel packet capturing. With these states one would have a better understanding how this solution can compete against a low-power system.

The project has only been tested on a single platform with a single Ethernet driver, hence it is not a generalized solution. However, testing different target boards would induce high costs and a tremendous effort to implement the necessary code and configure the platforms.

Moreover, the power measurement is not sufficiently accurate due to the limitations of the measurement device. Since professional power measurement devices are expensive, we decided not to purchase one and instead use a regular socket power meter. Because of that, we had to choose a more accurate measurement that this device presented - current. Ideally, one would use a high-grade measurement device to assess how much power our solution dissipates when conducting tests with various parameter values.

6.3 Future work

The idea to accelerate packet capturing in an embedded system was spontaneous and was chosen as the project topic after empirical study alone. In the future, it would be beneficial to have tools that propose which parts of the code could be offloaded to an external accelerator. Of course, this is a very complex topic, but since there are already HLS compilers, the way ahead is already being slowly opened.

6.3.1 What has been left undone?

Due to time shortages in the final stages of the project and non-trivial changes required to implement it, an interesting parameter is not tested - bus width. Setting this width to a greater value allows for transferring more data in a single burst and could have a big impact on performance and should be researched further.

Also, tests probing when an increase in burst length hinders the performance would have much scientific value for future implementors of similar solutions.

The XDP program used for steering the driver to offload the packets to FPGA only returns a value and does not perform any additional processing. Initially, it was meant to perform some early filtering of the packets and capture just the ones matching a particular filter. This had to be left undone as the scope of the project proved to be greater than expected.

Also, another functionality that could be done in the FPGA was not implemented, despite initial plans for the project to imitate the *tcpdump* program and support various filtering options that could be realized by the RTL code.

Testing the solution on a target board having low-power consumption capabilities would assert how does it fare against a CPU-only capturing which uses sleep states.

6.3.2 Next obvious things to be done

The bus width parametrization should be implemented for easier customization of the project and to add a degree of freedom for testing. It would also allow for a better performance during stress-testing the system with packet floods and potentially reduce power consumption.

Supporting a variable bus width in the module itself is equally important as transferring the solution to the FPGA2SDRAM bridge and debugging issues that prevented it from waking up. Moving to this bridge would remove the interconnect logic that was added to accommodate for differences between AXI and Avalon, reducing the logic requirements of the solution even further.

Since the project was written to be extensible, the RTL code allows for easy integrations of additional packet processing functionalities that could be done after the packet is received and before it is placed in the SDRAM buffer. An example of such functionality is packet filtering by classification or simple rule matching. Such implementations are not trivial but they are surely possible thanks to having a FIFO that essentially stores some data before it leaves the module.

As the project is modular, it could be used for egress packet capturing as well. Doing so would require some integrations in the kernel driver and a slightly different approach as to where hook the FPGA since the Linux kernel currently does not support egress XDP. No big changes would be necessary in the FPGA code though.

6.4 Reflections

As already mentioned in the section 1.8, every scientific project should contribute towards a better tomorrow. Even though this research concerns itself with a topic that is not directly related to human well-being, it still benefited the embedded systems computer science domain. Thanks to no total net increase in power consumption when running the FPGA code, the project proves that particular problems can and should be offloaded to hardware accelerators. As mentioned in 6.3, a step forward that would contribute towards environmental progress would be devising models capable of suggesting which parts of the code executed on the CPU could be accelerated. In the case of this project, this includes the creation of a processing solution that would decrease the power consumption of De0-Nano SoC. Even though this is a highly specialized solution, it proves that efficient utilization of a hardware accelerator is beneficial at times, both in performance and power consumption gains.

References

[1] Internet of Things statistics for 2022 - Taking Things Apart. Dataprot. [Online]. Available: https://dataprot.net/statistics/iot-statistics/ [Page 1.]

[2] R. Mahmoud, T. Yousuf, F. Aloul, and I. Zualkernan, "Internet of things (IoT) security: Current status, challenges and prospective measures," in *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, 2015. doi: 10.1109/ICITST.2015.7412116 pp. 336–341. [Page 1.]

[3] A. Pekar, J. Mocnej, W. K. G. Seah, and I. Zolotova, "Application Domain-Based Overview of IoT Network Traffic Characteristics," *ACM Comput. Surv.*, vol. 53, no. 4, jul 2020. doi: 10.1145/3399669. [Online]. Available: https://doi.org/10.1145/3399669 [Page 1.]

[4] Wireshark. Wireshark. [Online]. Available: https://www.wireshark.org/ [Page 1.]

[5] S. B. Alias, S. Manickam, and M. M. Kadhum, "A Study on Packet Capture Mechanisms in Real Time Network Traffic," in *2013 International Conference on Advanced Computer Science Applications and Technologies*, 2013. doi: 10.1109/ACSAT.2013.95 pp. 456–460. [Page 2.]

[6] tcpdump. The Tcpdump group. [Online]. Available: https://www.tcpdump.org/ [Page 2.]

[7] C. Cascaval, S. Chatterjee, H. Franke, K. J. Gildea, and P. Pattnaik, "A taxonomy of accelerator architectures and their programming models," *IBM J. Res. Dev.*, vol. 54, p. 5, 2010. [Page 3.]

[8] S. Lahti, M. Rintala, and T. D. Hämäläinen, "Leveraging Modern C++ in High-level Synthesis," *IEEE Transactions on Computer-Aided*

Design of Integrated Circuits and Systems, pp. 1–1, 2022. doi: 10.1109/TCAD.2022.3193646 [Pages 3 and 5.]

[9] E. Homsirikamol and K. Gaj, "Can high-level synthesis compete against a hand-written code in the cryptographic domain? a case study," in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, 2014. doi: 10.1109/ReConFig.2014.7032504 pp. 1–8. [Page 3.]

[10] T. Marc-André, "Two FPGA Case Studies Comparing High Level Synthesis and Manual HDL for HEP applications," 2018. [Online]. Available: https://arxiv.org/abs/1806.10672 [Page 3.]

[11] Netronome SmartNIC Overview. Netronome. [Online]. Available: https://www.netronome.com/products/smartnic/overview/ [Pages 3 and 16.]

[12] packagecloud. Illustrated Guide to Monitoring and Tuning the Linux Networking Stack: Receiving Data | Packagecloud Blog. packagecloud.io. [Online]. Available: https://blog.packagecloud.io/illustrated-guide-monitoring-tuning-linux-networking-stack-receiving-data/ [Page 4.]

[13] ——. Monitoring and Tuning the Linux Networking Stack: Receiving Data | Packagecloud Blog. packagecloud.io. [Online]. Available: https://blog.packagecloud.io/monitoring-tuning-linux-networking-stack-receiving-data/ [Page 4.]

[14] ——. Monitoring and Tuning the Linux Networking Stack: Sending Data | Packagecloud Blog. [Online]. Available: https://blog.packagecloud.io/monitoring-tuning-linux-networking-stack-sending-data/ [Page 4.]

[15] G. Borello. The art of writing eBPF programs: A primer. Sysdig. [Online]. Available: https://sysdig.com/blog/the-art-of-writing-ebpf-programs-a-primer/ [Page 4.]

[16] How to Receive a Million Packets per Second. The Cloudflare Blog. [Online]. Available: http://blog.cloudflare.com/how-to-receive-a-million-packets/ [Page 4.]

[17] BPF and XDP Reference Guide — Cilium 1.10.8 documentation. [Online]. Available: https://docs.cilium.io/en/v1.10/bpf/ [Page 4.]

[18] bcc - BPF compiler collection. iovisor. [Online]. Available: https://github.com/iovisor/bcc [Page 6.]

[19] A thorough introduction to bpftrace. Brendan Gregg. [Online]. Available: https://www.brendangregg.com/blog/2019-08-19/bpftrace.html [Page 6.]

[20] FPGA based hardware accelerator for musical synthesis for Linux system. Jakub Duchniewicz. [Online]. Available: https://jduchniewicz.com/FPGA-synth.pdf [Page 6.]

[21] W. Kim, S. Kim, and H. Lim, "Malicious Data Frame Injection Attack Without Seizing Association in IEEE 802.11 Wireless LANs," *IEEE Access*, vol. 9, pp. 16 649–16 660, 2021. doi: 10.1109/ACCESS.2021.3054130 [Page 10.]

[22] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, "Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels," in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, 2019. doi: 10.1109/ICESS.2019.8782524 pp. 1–8. [Page 11.]

[23] A. Ramaswami, T. Kenter, T. D. Kühne, and C. Plessl, "Evaluating the Design Space for Offloading 3D FFT Calculations to an FPGA for High-Performance Computing," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, S. Derrien, F. Hannig, P. C. Diniz, and D. Chillet, Eds. Cham: Springer International Publishing, 2021. ISBN 978-3-030-79025-7 pp. 285–294. [Pages 11 and 51.]

[24] R. Dick, G. Lakshminarayana, A. Raghunathan, and N. Jha, "Analysis of power dissipation in embedded systems using real-time operating systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 5, pp. 615–627, 2003. doi: 10.1109/TCAD.2003.810745 [Page 14.]

[25] A. Agarwal, S. Rajput, and A. S. Pandya, "Power Management System for Embedded RTOS: An Object Oriented Approach," in *2006 Canadian Conference on Electrical and Computer Engineering*, 2006. doi: 10.1109/CCECE.2006.277310 pp. 2305–2309. [Page 14.]

[26] F. Bennett, D. Clarke, J. Evans, A. Hopper, A. Jones, and D. Leask, "Piconet: embedded mobile networking," *IEEE Personal*

Communications, vol. 4, no. 5, pp. 8–15, 1997. doi: 10.1109/98.626977 [Page 14.]

[27] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, *TinyOS: An Operating System for Sensor Networks*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 115–148. ISBN 978-3-540-27139-0. [Online]. Available: https://doi.org/10.1007/3-540-27139-2_7 [Page 14.]

[28] Z. Shelby, P. Mahonen, J. Riihijarvi, O. Raivio, and P. Huuskonen, "NanoIP: the zen of embedded networking," in *IEEE International Conference on Communications, 2003. ICC '03.*, vol. 2, 2003. doi: 10.1109/ICC.2003.1204570 pp. 1218–1222 vol.2. [Page 14.]

[29] K. S. J. Pister, J. M. Kahn, and B. E. Boser, "Smart dust: Wireless networks of millimeter-scale sensor nodes." in *1999 Electronics Research Laboratory Research Summary*, 1999. [Page 14.]

[30] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating Systems for Low-End Devices in the Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 720–734, 2016. doi: 10.1109/JIOT.2015.2505901 [Page 14.]

[31] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, ser. LCN '04. USA: IEEE Computer Society, 2004. doi: 10.1109/LCN.2004.38. ISBN 0769522602 p. 455–462. [Online]. Available: https://doi.org/10.1109/LCN.2004.38 [Page 14.]

[32] R. Barry, *FreeRTOS reference manual: API functions and configuration options*. Real Time Engineers Limited, 2009. [Page 14.]

[33] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2013. doi: 10.1109/INFCOMW.2013.6970748 pp. 79–80. [Page 14.]

[34] Zephyr Project. The Linux Foundation. [Online]. Available: https://www.zephyrproject.org/ [Page 14.]

[35] J. J. Patoliya and M. M. Desai, "Face detection based ATM security system using embedded Linux platform," in *2017 2nd International Conference for Convergence in Technology (I2CT)*, 2017. doi: 10.1109/I2CT.2017.8226097 pp. 74–78. [Page 15.]

[36] M. Åsberg, T. Nolte, M. Joki, J. Hogbrink, and S. Siwani, "Fast Linux bootup using non-intrusive methods for predictable industrial embedded systems," in *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, 2013. doi: 10.1109/ETFA.2013.6648027 pp. 1–8. [Page 15.]

[37] H. Guo, Z. Wang, and X. Wang, "Transplant of Linux and Embedded System of Boot Loader and LED Driver," in *2010 International Conference on Machine Vision and Human-machine Interface*, 2010. doi: 10.1109/MVHI.2010.118 pp. 733–736. [Page 15.]

[38] D. Hart, J. Stultz, and T. Ts'o, "Real-time Linux in real time," *IBM Systems Journal*, vol. 47, no. 2, pp. 207–220, 2008. doi: 10.1147/sj.472.0207 [Page 15.]

[39] L.-C. Duca and A. Duca, "Achieving Hard Real-Time Networking on PREEMPT_RT Linux with RTnet," in *2020 International Symposium on Fundamentals of Electrical Engineering (ISFEE)*, 2020. doi: 10.1109/ISFEE51261.2020.9756165 pp. 1–4. [Page 15.]

[40] The Buildroot User Manual. [Online]. Available: https://buildroot.org/downloads/manual/manual.html#rebuild-pkg [Pages 15 and 38.]

[41] Yocto Project. Yocto Project. [Online]. Available: https://www.yoctoproject.org/ [Page 15.]

[42] Arduino MKR Vidor 4000. Arduino. [Online]. Available: https://store.arduino.cc/products/arduino-mkr-vidor-4000 [Page 16.]

[43] I.-S. Yoon, S.-H. Chung, and J.-S. Kim, "Implementation of Lightweight TCP/IP for Small, Wireless Embedded Systems," in *2009 International Conference on Advanced Information Networking and Applications*, 2009. doi: 10.1109/AINA.2009.53 pp. 965–970. [Page 16.]

[44] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, p. 263–297, aug 2000. doi: 10.1145/354871.354874. [Online]. Available: https://doi.org/10.1145/354871.354874 [Page 17.]

[45] DPDK. DPDK. [Online]. Available: https://www.dpdk.org/ [Page 17.]

[46] H. Zhang, Z. Chen, and Y. Yuan, "High-Performance UPF Design Based on DPDK," in *2021 IEEE 21st International Conference on Communication Technology (ICCT)*, 2021. doi: 10.1109/ICCT52962.2021.9657903 pp. 349–354. [Page 17.]

[47] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008. doi: 10.1109/JPROC.2008.917757 [Page 17.]

[48] NVIDIA, P. Vingelmann, and F. H. Fitzek, "CUDA," 2022. [Online]. Available: https://developer.nvidia.com/cuda-toolkit [Page 17.]

[49] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf [Page 17.]

[50] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283. [Page 17.]

[51] Google, "TPU," 2022. [Online]. Available: https://cloud.google.com/tpu/ [Page 17.]

[52] Khronos, "OpenCL," 2022. [Online]. Available: https://www.khronos.org/opencl/ [Page 17.]

[53] Y. Go, M. Jamshed, Y. Moon, C. Hwang, and K. Park, "APUNet: Revitalizing GPU as Packet Processing Accelerator," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17. USA: USENIX Association, 2017. ISBN 9781931971379 p. 83–96. [Pages 17 and 18.]

[54] GPGPU with GLES. Jakub Duchniewicz. [Online]. Available: https://github.com/JDuchniewicz/GPGPU-with-GLES [Page 17.]

[55] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-Accelerated Software Router," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, p. 195–206, aug 2010. doi: 10.1145/1851275.1851207. [Online]. Available: https://doi.org/10.1145/1851275.1851207 [Page 18.]

[56] A. Boutros and V. Betz, "FPGA Architecture: Principles and Progression," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 4–29, 2021. doi: 10.1109/MCAS.2021.3071607 [Page 18.]

[57] P. Possa, D. Schaillie, and C. Valderrama, "FPGA-based hardware acceleration: A CPU/accelerator interface exploration," in *2011 18th IEEE International Conference on Electronics, Circuits, and Systems*, 2011. doi: 10.1109/ICECS.2011.6122291 pp. 374–377. [Pages 18, 26, and 51.]

[58] F. O'Brien, M. Agostini, and T. S. Abdelrahman, "A Streaming Accelerator for Heterogeneous CPU-FPGA Processing of Graph Applications," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2021. doi: 10.1109/IPDPSW52791.2021.00014 pp. 26–35. [Pages 18 and 26.]

[59] FlexRAN. Intel. [Online]. Available: https://www.intel.com/content/www/us/en/developer/topic-technology/edge-5g/tools/flexran.html [Page 18.]

[60] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'18. USA: USENIX Association, 2018. ISBN 9781931971430 p. 51–64. [Page 18.]

[61] W. Jiang and V. K. Prasanna, "A FPGA-based Parallel Architecture for Scalable High-Speed Packet Classification," in *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2009. doi: 10.1109/ASAP.2009.17 pp. 24–31. [Page 18.]

[62] S. Zhou, Y. R. Qu, and V. K. Prasanna, "Large-scale packet classification on FPGA," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2015. doi: 10.1109/ASAP.2015.7245738 pp. 226–233. [Page 18.]

[63] M. S. Abdelfattah, A. Bitar, and V. Betz, "Design and Applications for Embedded Networks-on-Chip on FPGAs," *IEEE Transactions on Computers*, vol. 66, no. 6, pp. 1008–1021, 2017. doi: 10.1109/TC.2016.2621045 [Page 19.]

[64] M. Attig and G. Brebner, "400 Gb/s Programmable Packet Parsing on a Single FPGA," in *2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, 2011. doi: 10.1109/ANCS.2011.12 pp. 12–23. [Page 19.]

[65] D. Cerović, V. Del Piccolo, A. Amamou, K. Haddadou, and G. Pujolle, "Fast Packet Processing: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 3645–3676, 2018. doi: 10.1109/COMST.2018.2851072 [Page 19.]

[66] R. Duarte, C. Liu, and X. Niu, "RSA Cryptography Acceleration for Embedded System," 2010. [Page 19.]

[67] C.-L. Su, C.-Y. Tsui, and A. Despain, "Saving power in the control path of embedded processors," *IEEE Design & Test of Computers*, vol. 11, no. 4, pp. 24–31, 1994. doi: 10.1109/54.329448 [Page 19.]

[68] B. Moyer, "Low-power design for embedded processors," *Proceedings of the IEEE*, vol. 89, no. 11, pp. 1576–1587, 2001. doi: 10.1109/5.964439 [Page 19.]

[69] N. Maruyama, T. Ishihara, and H. Yasuura, "An RTOS in hardware for energy efficient software-based TCP/IP processing," *Application Specific Processors, Symposium on*, vol. 0, pp. 58–63, 06 2010. doi: 10.1109/SASP.2010.5521147 [Page 19.]

[70] T. Gomes, F. Salgado, S. Pinto, J. Cabral, and A. Tavares, "Towards an FPGA-based network layer filter for the Internet of Things edge devices," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016. doi: 10.1109/ETFA.2016.7733684 pp. 1–4. [Page 20.]

[71] M. Silva, T. Gomes, and S. Pinto, "Agnostic Hardware-Accelerated Operating System for Low-End IoT," 08 2022. doi: 10.1109/RTCSA55878.2022.00009 [Page 20.]

[72] P. P. Czapski and A. Sluzek, "Experiments on data processing algorithms: Energy efficiency of wireless and untethered Field Programmable Gate Array (FPGA)-based embedded systems," in *2008 International Conference on Electronic Design*, 2008. doi: 10.1109/ICED.2008.4786635 pp. 1–8. [Page 20.]

[73] X. Ding and J. Wu, "Study on Energy Consumption Optimization Scheduling for Internet of Things," *IEEE Access*, vol. 7, pp. 70 574–70 583, 2019. doi: 10.1109/ACCESS.2019.2919769 [Page 20.]

[74] N. Paulino, J. a. C. Ferreira, and J. a. M. P. Cardoso, "Improving Performance and Energy Consumption in Embedded Systems via Binary Acceleration: A Survey," *ACM Comput. Surv.*, vol. 53, no. 1, feb 2020. doi: 10.1145/3369764. [Online]. Available: https://doi.org/10.1145/3369764 [Page 20.]

[75] PCAP Capture File Format. The tcpdump group. [Online]. Available: https://www.ietf.org/archive/id/draft-gharris-opsawg-pcap-01.html [Page 20.]

[76] libpcap. The tcpdump group. [Online]. Available: https://github.com/the-tcpdump-group/libpcap [Page 20.]

[77] Altera De0-Nano SoC GHRD. Altera. [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=941 [Page 31.]

[78] Avalon MM Specification. Intel. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683091/20-1/introduction-to-the-interface-specifications.html [Page 32.]

[79] Building a Skid Buffer for AXI processing. The ZipCPU by Gisselquist Technology. [Online]. Available: https://zipcpu.com/blog/2019/05/22/skidbuffer.html [Page 36.]

[80] uboot fork for SoCFPGA. Altera. [Online]. Available: https://github.com/altera-opensource/u-boot-socfpga [Page 38.]

[81] Linux Kernel fork for SoCFPGA. Altera. [Online]. Available: https://github.com/altera-opensource/linux-socfpga [Page 38.]

Appendix A

Major obstacles faced

No project is without its issues and time-consuming nasty bugs. In the case of this project, probably the most irksome and time-wasting was our inability to get the `fpga2sdram` bridges up and running. Having perused every resource available on the internet up to this day, invoked bootloader scripts that would set proper values in registers, and even switched software versions to accommodate for any possible changes we still were not able to get it running. This created a major delay after which we decided to use regular bridges using the AXI protocol and lose some width (256 bits to 128 reduction). After this change, we were finally able to see the data being transferred between FPGA and RAM.

We also encountered some bugs in our initial erroneous implementation of write control where `write` and `waitrequest` signals could come at various times and cause unexpected states to be present in our logic. This was alleviated by implementing a counter for packet transmission in addition to regular burst segment counting logic. Changes in this area caused yet another bug with hanging bus and board hanging in seemingly unrelated places. It turned out that the interconnect would sometimes erroneously translate our burst count requests, causing the bus to wait on a pending transfer when the FPGA code was already done. Since the bus controller was waiting for a transaction it blocked other accesses to SDRAM and due to popular demand for this peripheral, other components of the system crashed with timeouts. A solution to this problem was switching from specifying burst lengths in bytes to words.

Some other major time-consuming issues were related to getting the eBPF support on the board alongside the necessary userspace applications that were capable of loading the XDP programs. Choosing a proper kernel version was

crucial, as XDP support was added only after kernel 4.8 and as mentioned previously efficient execution of these programs requires JIT compilation which also requires a fairly new kernel. Additionally, it turned out that we had to add a newer version of `iptables2` than Buildroot initially provided that we ultimately chose for loading the XDP program.

Finally, understanding the Ethernet kernel driver to discover which parts have most relevance to our problem, and properly integrating our code proved to be quite a demanding task. We spent significant time making sense why packets would not be passed from kernel to the FPGA only to discover that the addresses that arrive along with the `xdp` struct were virtual and not physical.

Appendix B

Write control code listing

```
module wr_ctrl
                #(parameter BURST_SIZE_WORDS = 4)
                (input logic clk,
                input logic reset,
                input logic wr_ctrl,
                input logic empty,
                input logic [31:0] fifo_out,
                output logic rd_from_fifo,
                output logic wr_ctrl_rdy,
                input logic [31:0] control,
                input logic [31:0] pkt_begin,
                input logic [31:0] pkt_end,
                input logic [31:0] capt_buf_start,
                input logic [31:0] capt_buf_size,
                input logic [8:0] usedw,
                input logic [31:0] seconds,
                input logic [31:0] nanoseconds,
                output logic [31:0] last_write_addr,
                output logic capt_buf_wrap,
                // avalon (host)master signals
                output logic [31:0] address,
                output logic [31:0] writedata,
                output logic write,
                output logic [15:0] burstcount,
                input logic waitrequest
            );

enum logic [2:0] { IDLE, PREP, WR_TIMESTAMP, WR_PKT_DATA,
    DONE } state, state_next;

logic [31:0] reg_control, reg_pkt_begin, reg_pkt_end,
```

```
31              reg_capt_buf_start, reg_capt_buf_size;
32 logic done_reading, start_transfer, first_transaction;
33
34 logic [15:0] total_burst_remaining,
35              burst_segment_remaining_count,
36              total_size, BYTES_IN_BURST,
37              burstsize_in_words, WORD_SIZE;
38
39 logic [31:0] bytes_to_buf_end;
40
41 logic [31:0] capt_buf_end;
42
43 logic [15:0] burst_size;
44 logic burst_start, burst_end, first_burst_wait_fifo_fill;
45 logic skbf1_valid, skbf2_valid, tx_accept, skbf1_ready,
       skbf2_ready;
46 logic [31:0] timestamp_pkt_reg;
47
48 logic [31:0] int_address, int_writedata, fifo_out_d;
49 logic [15:0] int_burstcount;
50 logic int_write;
51 logic [79:0] skbf1_in_data, skbf2_in_data, skbf2_out_data;
52 logic skbf1_data_valid;
53
54 logic [2:0] timestamp_pkt_cnt;
55
56 logic [15:0] tx_accept_counter;
57
58 assign WORD_SIZE = 'd4;
59 assign BYTES_IN_BURST = BURST_SIZE_WORDS * WORD_SIZE;
60
61 // constant assignments for bytes to words conversion
       required by
62 // interconnect
63 assign capt_buf_end = reg_capt_buf_start + reg_capt_buf_size;
64 assign bytes_to_buf_end = first_transaction ? capt_buf_end -
       capt_buf_start : capt_buf_end - last_write_addr;
65
66 assign burstsize_in_words = burst_size[15:2] + (burst_size
       [1:0] !== 'h0); // $ceil(burst_size/4.0)
67
68 assign total_size = (reg_pkt_end - reg_pkt_begin);
69
70 assign tx_accept = write && !waitrequest;
71
72 assign timestamp_pkt_reg = (timestamp_pkt_cnt == 'd4) ?
       seconds :
```

```
                                ((timestamp_pkt_cnt == 'd3) ?
    nanoseconds : total_size);

assign skbf1_in_data[79:32] = { int_address, int_burstcount
    };
assign skbf1_in_data[31:0] = (state == WR_TIMESTAMP) ?
    timestamp_pkt_reg : int_writedata;

assign skbf1_data_valid = (state == WR_TIMESTAMP) ?
    timestamp_pkt_cnt !== '0 : int_write;

assign int_writedata = skbf1_ready ? fifo_out : fifo_out_d;

// Avalon MM interface signals
assign write = skbf2_valid;
assign address = skbf2_out_data[79:48];
assign burstcount = skbf2_out_data[47:32];
assign writedata = skbf2_out_data[31:0];

assign wr_ctrl_rdy = done_reading;

skidbuffer #(
    .DW(80),
    .OPT_INITIAL(0),
    .OPT_OUTREG(0)
)
skbf1 (
    .i_clk(clk),
    .i_reset(~reset),
    // Left
    .i_valid(skbf1_data_valid),
    .o_ready(skbf1_ready),
    .i_data(skbf1_in_data),
    // Right
    .o_valid(skbf1_valid),
    .i_ready(skbf2_ready),
    .o_data(skbf2_in_data)
),

skbf2 (
    .i_clk(clk),
    .i_reset(~reset),
    // Left
    .i_valid(skbf1_valid),
    .o_ready(skbf2_ready),
    .i_data(skbf2_in_data),
    // Right
```

```
    .o_valid(skbf2_valid),
    .i_ready(!waitrequest),
    .o_data(skbf2_out_data)
);


always_ff @(posedge clk) begin : states
    if (!reset) begin
        state <= IDLE;
    end
    else begin
        state <= state_next;
    end
end

always_comb begin : fsm
    case (state)
        IDLE:   begin
                if (wr_ctrl) begin
                    state_next = PREP;
                end
                else begin
                    state_next = IDLE;
                end
                end

        PREP:   begin
                    state_next = WR_TIMESTAMP;
                end

WR_TIMESTAMP:   begin
                if (burst_end && timestamp_pkt_cnt == '0)
   begin
                    state_next = WR_PKT_DATA;
                end
                else begin
                    state_next = WR_TIMESTAMP;
                end
                end

 WR_PKT_DATA:   begin
                if (done_reading) begin
                    state_next = DONE;
                end
                else begin
                    state_next = WR_PKT_DATA;
                end
```

```
162                 end
163
164         DONE:   begin
165                     state_next = IDLE;
166                 end
167     endcase
168 end
169
170 always_ff @(posedge clk) begin : avalon_mm_ctrl
171     if (!reset) begin
172         int_address <= '0;
173         int_burstcount <= '0;
174         int_write <= '0;
175         first_transaction <= '1; // should be done on
    register writing
176         capt_buf_wrap <= 'b0;
177     end
178     else begin
179         if (start_transfer && first_transaction) begin
180         // this is start of every packet, it will overwrite,
181         // flag for resetting last_write_addr
182             int_address <= reg_capt_buf_start;
183             first_transaction <= '0;
184         end
185         else if (burst_end) begin
186             if (int_address + burst_size >= capt_buf_end)
    begin
187                 int_address <= reg_capt_buf_start;
188                 capt_buf_wrap <= 'b1;
189             end
190             else begin
191                 int_address <= int_address + burst_size;
192             end
193             if (total_burst_remaining > burst_size) begin
194                 if (int_address + 2 * burst_size >=
    capt_buf_end) begin
195                     last_write_addr <= reg_capt_buf_start;
196                 end
197                 else begin
198                     last_write_addr <= int_address + 2 *
    burst_size;
199                 end
200             end
201             else begin
202                 last_write_addr <= int_address + burst_size;
203             end
204         end
```

```
205
206         if (start_transfer) begin // TODO: check if proper
207             int_burstcount <= 'h4; // timestamp
208         end
209         else if (burst_start) begin
210             int_burstcount <= burstsize_in_words;
211         end
212
213         if (state == WR_PKT_DATA) begin
214             if (rd_from_fifo) begin
215             // arm int_write if there was a FIFO read
216             // in previous CC (data to be sent)
217                 int_write <= 'b1;
218             end
219             else if (skbf1_ready) begin
220             // clear int_write if the last word
221             // has been accepted by skbf1
222             // (rd_from_fifo is already down)
223                 int_write <= 'b0;
224             end
225         end else begin
226             int_write <= 'b0;
227         end
228     end
229 end
230
231 always_ff @(posedge clk) begin : start_ctrl
232     if (!reset) begin
233         reg_control <= '0;
234         reg_pkt_begin <= '0;
235         reg_pkt_end <= '0;
236         reg_capt_buf_start <= '0;
237         reg_capt_buf_size <= '0;
238     end
239
240     start_transfer <= 'b0;
241
242     if (state == IDLE && state_next == PREP) begin
243         start_transfer <= 'b1;
244         reg_control <= control;
245         reg_pkt_begin <= pkt_begin;
246         reg_pkt_end <= pkt_end;
247         reg_capt_buf_start <= capt_buf_start;
248         reg_capt_buf_size <= capt_buf_size;
249     end
250 end
251
```

```
252 always_ff @(posedge clk) begin : avalon_mm_tx
253     if (!reset) begin
254         first_burst_wait_fifo_fill <= 'b0;
255         total_burst_remaining <= '0;
256         burst_segment_remaining_count <= '0;
257         timestamp_pkt_cnt <= '0;
258         tx_accept_counter <= '0;
259     end
260     else begin
261 /***************************/
262 /* Set the total burst length
263     at the start of a packet
264 /***************************/
265         if (start_transfer) begin
266             total_burst_remaining <= total_size + 'd16;
267             first_burst_wait_fifo_fill <= 'b1;
268             timestamp_pkt_cnt <= 'd4; // seconds, nanoseconds
    , pkt_len x2
269         end
270         else if (burst_end) begin
271             total_burst_remaining <= total_burst_remaining -
272                     (total_burst_remaining < BYTES_IN_BURST ?
273                     total_burst_remaining : burst_size);
274         end
275 
276 /*****************************/
277 /* Mark the start of a burst */
278 /*****************************/
279         burst_start <= 'b0;
280 
281         if (start_transfer) begin
282             burst_start <= 'b1;
283             burst_size <= (bytes_to_buf_end < BYTES_IN_BURST
    ?
284                            bytes_to_buf_end : BYTES_IN_BURST
    );
285                            // first burst is a timestamp
286         end
287 
288         if (state == WR_TIMESTAMP &&
289             burst_end &&
290             timestamp_pkt_cnt > '0) begin // don't mix with
    data
291             burst_start <= 'b1;
292             burst_size <= timestamp_pkt_cnt * WORD_SIZE;
293         end
294         else if ((state == WR_PKT_DATA &&
```

```
295                 (first_burst_wait_fifo_fill && usedw >=
    BURST_SIZE_WORDS)) ||
296                 (!first_burst_wait_fifo_fill && burst_end &&
297                 total_burst_remaining > BYTES_IN_BURST))
    begin
298             burst_start <= 'b1;
299             first_burst_wait_fifo_fill <= 'b0;
300 
301             if (bytes_to_buf_end < BYTES_IN_BURST) begin
302                 burst_size <= (total_burst_remaining -
    BYTES_IN_BURST < bytes_to_buf_end) ?
303                                 total_burst_remaining -
    BYTES_IN_BURST : bytes_to_buf_end;
304             end
305             else begin
306                 burst_size <= (total_burst_remaining -
    BYTES_IN_BURST < BYTES_IN_BURST) ?
307                                 total_burst_remaining -
    BYTES_IN_BURST : BYTES_IN_BURST;
308             end
309         end
310 
311 /*************************/
312 /* Mark the end of a burst */
313 /*************************/
314         burst_end <= 'b0;
315 
316         if (tx_accept_counter <= 'h4
317             && tx_accept_counter > 'h0
318             && tx_accept) begin
319             burst_end <= 'b1;
320         end
321 
322 /*********************************************************
    */
323 /* Count the number of timestamp words put into write stage
    */
324 /*********************************************************
    */
325         if (state == WR_TIMESTAMP && skbf1_ready &&
326             skbf1_data_valid && timestamp_pkt_cnt != 'd0)
    begin
327             timestamp_pkt_cnt <= timestamp_pkt_cnt - 'b1;
328         end
329 
330 /***********************************************/
331 /* Control the number of packet bytes
```

```
332    put into write stage within the current burst
333 /*************************************************/
334         if (burst_start) begin
335             burst_segment_remaining_count <= burst_size;
336         end
337         else if ((state == WR_TIMESTAMP && skbf1_ready)
338                 || (state == WR_PKT_DATA && rd_from_fifo))
    begin
339             if (burst_segment_remaining_count > 'h0) begin
340                 if (burst_segment_remaining_count < 'h4)
    begin
341                     burst_segment_remaining_count <= '0;
342                 end
343                 else begin
344                     burst_segment_remaining_count <=
345                         burst_segment_remaining_count - 'h4;
346                 end
347             end
348         end
349
350 /***********************************/
351 /* Control the number of transmitted
352 /* bytes within the current burst
353 /***********************************/
354         if (burst_start) begin
355             tx_accept_counter <= burst_size;
356         end
357         else if (tx_accept && tx_accept_counter > '0) begin
358             if (tx_accept_counter < 'h4) begin
359                 tx_accept_counter <= '0;
360             end
361             else begin
362                 tx_accept_counter <= tx_accept_counter - 'h4;
363             end
364         end
365
366 /*****************************/
367 /* Mark the end of a packet */
368 /*****************************/
369         done_reading <= 'b0;
370
371         // just trigger it for one cycle
372         if (total_burst_remaining <= BYTES_IN_BURST &&
373             tx_accept_counter === 0 && burst_end) begin
374             done_reading <= 'b1;
375         end
376
```

```
377 /*************************************/
378 /* We need to retain last FIFO output in
379 /* case there is a stall at skbf1 input
380 /*************************************/
381         fifo_out_d <= fifo_out;
382     end
383 end
384 
385 always_comb begin : fifo_ctrl
386     rd_from_fifo <= '0;
387 
388     if (state == WR_PKT_DATA && burst_segment_remaining_count
    > '0
389         && !empty && skbf1_ready) begin
390         rd_from_fifo <= 'b1;
391     end
392 end
393 endmodule
```

Listing B.1: SystemVerilog code for the write control module.

TRITA-EECS-EX- 2022:00

€€€€ For DIVA €€€€

{
"Author1": { ”Last name”: ”Duchniewicz”,
"First name": ”Jakub”,
"E-mail": ”jakubdu@kth.se”,
"organisation": {”L1”: ”School of Electrical Engineering and Computer Science”,
}
},
"Cycle": ”2”,
"Course code": ”DA258X”,
"Credits": ”30.0”,
"Degree1": {”Educational program”: ”Master’s Programme, ICT Innovation, 120 credits”
,”programcode”: ”TIVNM”
,”Degree”: ”Masters degree”
,”subjectArea”: ”Embedded Systems”
},
"Title": {
"Main title": ”FPGA accelerated packet capture with eBPF”,
"Subtitle": ”Performance considerations of using SoC FPGA accelerators for packet capturing.”,
"Language": ”eng” },
"Alternative title": {
"Main title": ”FPGA-accelererad paketfångst med eBPF”,
"Subtitle": ”Prestandaöverväganden vid användning av SoC FPGA acceleratorer för paketering.”,
"Language": ”swe”
},
"Supervisor1": { ”Last name”: ”Pisklak”,
"First name": ”Sebastian”,
"Local User Id": ”u100003”,
"E-mail": ”sebastian.pisklak@tietoevry.com”,
"organisation": {”L1”: ”School of Electrical Engineering and Computer Science”,
"L2": ”Computer Science” }
},
"Supervisor2": { ”Last name”: ”Thilanka Thilakasiri Laddusinghe Badu”,
"First name": ”Hasini”,
"E-mail": ”thilanka@kth.se”,
},
"Examiner1": { ”Last name”: ”Becker”,
"First name": ”Matthias”,
"Local User Id": ”u1d13i2c”,
"E-mail": ”mabecker@kth.se”,
"organisation": {”L1”: ”School of Electrical Engineering and Computer Science”,
"L2": ”Computer Science” }
},
"Cooperation": { ”Partner_name”: ”Tietoevry”},
"National Subject Categories": "10201, 20207",
"Other information": {”Year”: ”2022”, ”Number of pages”: ”1,79”},
"Series": { ”Title of series”: ”TRITA-EECS-EX” , ”No. in series”: ”2022:00” },
"Opponents": { ”Name”: ”L. M. Waller & C. Zhexin”},
"Presentation": { ”Date”: ”2022-03-15 13:00”
,”Language”:”eng”
,”Room”: ”via Zoom https://kth-se.zoom.us/j/ddddddddddd”
,”Address”: ”Isafjordsgatan 22 (Kistagången 16)”
,”City”: ”Stockholm” },
"Number of lang instances": "3",
"Abstract[eng]": €€€€

With the rise of the Internet of Things and the proliferation of embedded devices equipped with an accelerator arose a need for efficient resource utilization. Hardware acceleration is a complex topic that requires specialized domain knowledge about the platform and different trade-offs that have to be made, especially in the area of power consumption. Efficient work offloading strives to reduce or at least maintain the total power consumption of the system. Offloading packet capturing is usually done in more powerful devices, hence scarce research is present concerning network packet acceleration in embedded devices.

The thesis focuses on accelerating networking packets utilizing a Field Programmable Gate Array in an embedded Linux System. The solution is based on a custom Linux distribution assembled using the Buildroot tool, specially configured and patched Linux kernel, uboot bootloader, and the programmable logic for packet acceleration. The system is evaluated on a De0-Nano System on Chip development board through modifications to burst lengths, packet sizes, and programmable logic clock frequency. Metrics include packet capturing time, time per packet, and consumed power. Finally, the results are contrasted with baseline embedded Linux packet processing by inspection of a packet’s path through the kernel.

Collected results provide a deeper understanding of the packet acceleration problem in embedded devices and the resultant system gives a solid starting point for possible extensions such as packet filtering. Key findings include an improvement in packet processing speed as the clock frequency and burst length are increased while maintaining power consumption. Additionally, the solution performs better when the packet sizes are above 64 bytes as the overhead of additional logic necessary for

their processing is compensated. The project is also found to be significantly faster than regular in kernel processing with the caveat of providing just packet capturing whereas Linux contains a full network stack.

€€€€,
"Keywords[eng]": €€€€
Field Programmable Gate Array, Acceleration, Networking, Embedded Linux €€€€,
"Abstract[swe]": €€€€

I och med uppkomsten av sakernas internet och spridningen av inbyggda enheter som är utrustade med en accelerator har det uppstått ett behov av effektivt resursutnyttjande. Hårdvaruacceleration är ett komplext ämne som kräver specialiserad domänkunskap om plattformen och olika avvägningar som måste göras, särskilt när det gäller energiförbrukning. Effektiv arbetsavlastning strävar efter att minska eller åtminstone bibehålla systemets totala energiförbrukning. Avlastning av paketering sker vanligtvis i kraftfullare enheter, och därför finns det knappt någon forskning om nätverksacceleration av paket i inbyggda enheter.

Avhandlingen är inriktad på att påskynda nätverkspaket med hjälp av en Field Programmable Gate Array i ett inbäddat Linuxsystem. Lösningen bygger på en anpassad Linuxdistribution som sammanställts med hjälp av verktyget Buildroot, en särskilt konfigurerad och patchad Linuxkärna, uboot bootloader och den programmerbara logiken för paketacceleration. Systemet utvärderas på ett De0-Nano System on Chip-utvecklingskort genom ändringar av burstlängder, paketstorlekar och den programmerbara logikens klockfrekvens. Metrikerna omfattar tid för paketering, tid per paket och förbrukad effekt. Slutligen jämförs resultaten med grundläggande inbäddad Linux-paketbehandling genom inspektion av paketens väg genom kärnan.

De samlade resultaten ger en djupare förståelse för problemet med paketacceleration i inbyggda enheter och det resulterande systemet ger en solid utgångspunkt för möjliga utvidgningar, t.ex. paketfiltrering. Bland de viktigaste resultaten kan nämnas en förbättring av hastigheten i paketbehandlingen när klockfrekvensen och burstlängden ökas samtidigt som strömförbrukningen bibehålls. Dessutom fungerar lösningen bättre när paketstorleken är större än 64 bytes eftersom den extra logik som krävs för att behandla paketen kompenseras. Projektet har också visat sig vara betydligt snabbare än vanlig kärnbearbetning, med den reservationen att det bara tillhandahåller paketupptagning, medan Linux innehåller en fullständig nätverksstack.

€€€€,
"Keywords[swe]": €€€€
Field Programmable Gate Array, Acceleration, Nätverksarbete, Inbyggd Linux €€€€,
"Abstract[pl]": €€€€

Rozwój Internetu Rzeczy i ąrosnca śćpopularno systemów wbudowanych ąposiadajcych wbudowany akcelerator ęsprztowy łsprawiy, że łwzrosa potrzeba na ich efektywne wykorzytanie. Akceleracja ęsprztowa jest ądziedzin nauki, która wymaga specjalistycznej wiedzy na temat platformy na której ma ćoperowa oraz wymaga śznajomoci potencjalnych komplikacji które ęsi z ąni ążąwi. Efektywna akceleracja ma na celu ęredukcj żzuycia energii, a przynajmnniej jej utrzymanie na dotychczasowym poziomie. Tematyka ta jest śćdo uboga pod ąktem ędostpnej literatury, żgdy zazwyczaj akceleratory stosowane do sieciowych ąńrozwiza ąs żuywane w ąrozwizaniach serwerowych gdzie ęąwystpuj innego rodzaju problemy.

W pracy wykorzystany jest akcelerator Field Programmable Gate Array który jest ęśączczi łpytki deweloperskiej De0-Nano System on Chip, gdzie łdziaa łąwspópracujc z wbudowanym systemem Linux, do którego przygotowania wykorzystano ęnarzdzie Buildroot. Na ńkocowe ąrozwizanie ponadto łskada ęsi łpoatane ąjdro Linuxa, bootloader uboot oraz programowalna logika ąrealizujca przechwytywanie pakietów sieciowych. ąRozwizanie poddane jest testom, w których parametry odpowiedzialne za łśćdugo transakcji typu burst, rozmiaru pakietu oraz ęśczstotliwoci zegara ąs poddawane modyfikacjom. Wyniki ąs przedstawione za ąpomoc czasu przetwarzania pakietu, czasu per pakiet oraz żzuycia mocy. Do oceny śefektywnoci ąrozwizania łżłposuyo żtake porównanie z czasem procesowania pakietu w niezmodyfikowanym systemie Linux

Na podstawie eksperymentów dokonanych w pracy ęwysunite ąs ęąnastpujce wnioski: wraz ze wzrostem ęśczstotliwoci zegara oraz łśdugoci transakcji burst, czas procesowania pakietów maleje a żzuycie ąprdu pozostaje na dotychczasowym poziomie. Pakiety o rozmiarze ąprzekraczajcym 64 bajty ąs procesowane wydajniej w dostarczonym ąrozwizaniu poprzez ękompensacj dodatkowego łnakadu czasu narzuconego przez ęlogik ąąązarzdzajc przetwarzaniem. System porównano żtake do łzwykego przetwarzania pakietów ąodbywajcego ęsi w systemie Linux które łokazao ęsi zdecydowanie wolniejsze z żzastrzeeniem, żi ów system dokonuje łpenego przetworzenia pakietów a ąrozwizanie w pracy jedynie ich przechwytywania. Projekt stanowi ępodstaw do ewentualnych ńrozszerze, na łprzykad filtrowania pakietów. Wnioski ęwysunite łżąsu łępogbieniu wiedzy w domenie sieci wbudowanych systemów Linux oraz ęsprztowej akceleracji.

€€€€,
"Keywords[pl]": €€€€
Field Programmable Gate Array, sprzętowa akceleracja, sieci internetowe, wbudowany system Linux €€€€,
}