Report

High performance processors and systems project

Edoardo Carlotto

edoardo.carlotto@mail.polimi.it
Pers. code 10730627
Matr. 242673

prof. Marco Domenico Santambrogio
a.y. 2023-2024

Contents

1 Introduction

1.1 Context

The network interface controller (NIC) is the gateway through which a computer interacts with the network. The NIC forms a bridge between the software stack and the network, and the functions of this bridge define the network interface. The functions of the network interface, as well as the implementation of those functions, are both evolving rapidly. This evolution is mainly driven by the dual requirements of increasing line rates and NIC features that support high-performance distributed computing and virtualization. In order to offer such features with increasing line rates, many NIC functions must be implemented in hardware. Common tasks previously implemented in software (e.g. deal with packets segmentation) can be performed with FPGAs in hardware, this is called hardware offloading. Also, with this offloads available, it is possible to use more efficient network I/O libraries and interfaces, including DPDK and XDP, capable of using FPGA resources at best. Therefore, by exploiting offloads on NIC, applications can reduce processing done with the CPU. NICs integrated with an FPGA, able to offer these features, are called SmartNICs. Even using these devices, utilizing 100 Gbps links fully exploiting the bandwidth remains challenging. Also, Microsoft researchers show that FPGAs are the best current platform for offloading network computations [1]. The authors of Ensō [2] demonstrate some of the challenges are due to inefficiencies in how the software communicates with the NIC.

The interface most NICs provide relies on the exchange of fixed-size buffers between the card and the host. The size is chosen by the software and usually is large enought to fit packets as big as the Maximum Transmission Unit (MTU), those are the largest that can be communicated in a single network layer transaction. Hardware offloading works on inputs that can span multiple packets and vary in size. Consequently, the software must split and recombine data into multiple buffers to communicate with the NIC, this process is inefficient and leads to performance bottlenecks. Moreover, the effectiveness of several I/O and CPU optimization is reduced: when the software sends batches of packets to be processed to the hardware, there is no guarantee that such packets are in contiguous and sequential memory locations. In addition to these motivations, it is worth mentioning that if the packets exchanged between the endpoints are not MTU-sized but tend to be smaller, such inefficiencies are exacerbated by fixed cost of descriptors. Descriptors hold metadata, including packet size, what processing should the NIC perform, a flag bit and a pointer to the buffer actually containing the data. Figure 1, taken from the Ensō paper [2], shows PCIe bandwidth utilization when transferring small packets: up to 39% of read bandwidth is consumed with descriptors. The authors of the paper explain that this happens with 64-bytes transfers and leads to a drop of the theoretical maximum throughput of PCIe to only 85 Gbps.

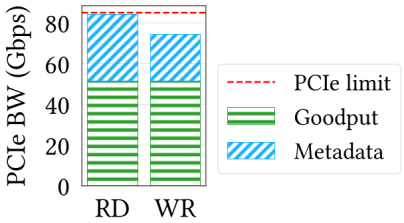


Figure 1. PCIe bandwidth utilization

Any kind of interface that relies of fixed-size buffers will have to deal with this challenge.
The authors of Ensō [2] propose a new streaming interface to tackle with these limitations capable of operating at line rates up to 100 Gbps. Unfortunately, at the time of writing Ensō requires an Intel Stratix 10 MX FPGA and that is the only supported board, keeping out different models and FPGA vendors from its benefits.

1.2 Motivations

The goal of this research project is to analyse Ensō to understand how it can be ported on other boards, such as the Xilinx Alveo AU250. The task at hand is an explorative one, as the source code is huge and it was not possible to understand a-priori whether it was feasible to port the code without rewriting it completely. It was also important to focus on the methodology to render the porting feasible, developing a viable strategy. Additionaly, the module developed is written using High Level Syntesys code and integrates with existing code in Hardware Description Language. Remarkably, Ensō offers application level primitives, called using a C++ library, that are used by applications to interact directly with the NIC. This implies that porting Ensō on a different board would require to adapt both the Hardware Description Language code and the C++ library to fully support the new environment. For this research project the focus will mainly be on the former, leaving a more in-depth analysis of the C++ library for future work.
First of all, it was needed to understand the features offered by the different projects available in the research community, and to dive into the details of Enso, too. This aimed at finding out what already available solutions could offer and to extract knowledge from the design decisions made in similar contexts. The goal was to find a project suitable for integrating Ensō, without having to redo something that was already available. Hardware-accelerated network-attached applications have very different requirements depending on their specific context; therefore products and research results aimed at targeting this field vary a lot in characteristics and features offered. It is common to find a solution that addresses a specific use case and

it is not easily adaptable to other applications, even if those differ only slightly from the intended purpuse. To make the best out of this environment, a state of the art analysis was mandatory. This study was performed to understand the Ensō key features and consequently to evaluate all the available candidates. It is important to mention that the source code of Ensō is huge and, as it frequently happens in the networking field, its logic involves working at different levels of abstraction. This means that the code is not easy to understand and that the task of porting it is not trivial. The main result of the state of the art analysis lead to chosing a solution, OpenNIC, supporting the integration of custom logic allowed to access the data path. This was intended to be used to port Ensō.
After this choice, the next important step was to design an interface to allow the integration of the ported code with OpenNIC. Such task involves being able to adapt a project interacting with applications on one side and defining hardware components on the other. This, coupled with the fact that the logic of Ensō is meant to introduce a different communication interface, is what makes the porting task complicated and challenging. This research project shows the feasibility of the integration explained above, by developing a demo to prove the concept.

2 State of the art

2.1 Ensō

The Ensō [2] work is proposing a new communication abstraction, in order to allow the software to interact with the NIC without a packetized interface in the middle. The communication model based on fixed-size packets has different limitations: those are mainly related to the inefficiency of splitting and recombining data and the fixed cost of metadata even with smaller packets; these issues where explained in the introduction section. The proposed abstraction is based on offering software communication as a stream of arbitrary data size. Applications can directly interact with Ensō through a C++ library; in this way the NIC is able to interact at application layer. Providing such benefits heavily relies on the concept of Ensō pipes: unbounded buffers to exchange data, requested using library calls. Ensō pipes are managed by the library and then mapped directly to data ring buffers in the memory of the FPGA, which contain directly the application data without metadata. To achieve the advertised performance, Ensō also implements an efficient notification mechanism to properly coordinate pipes between the CPU and the FPGA.
In addition to these main aspects, Ensō also offers other features. As the whole approach is based on making no assumption abount the data being handled, the stream abstraction can be repurposed depending on the application, without any additional cost or integration. Ensō can work at different levels, it could be employed as a traditional NIC providing simple offloading capabilites, it could offer transport offloading handling application-level messages and can be integrated with the application logic to handle application-level data directly. The key driver of SmartNICs nowadays is being able to work at high line rates, Ensō is not only capable of operating up to 100 Gbps but is also able to handle such high bandwidth transfer when network packets are significantly smaller than the MTU size.

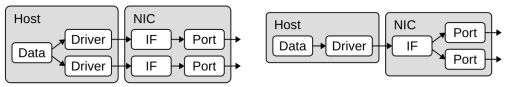
2.2 OpenNIC

AMD's OpenNIC project consists of three components: a NIC shell, a Linux kernel driver and a DPDK driver. The goal of OpenNIC is to enable fast prototyping of hardware-accelerated network-attached applications. Its documentation states that it is not a fully-fledged SmartNIC solution: there are still features not supported as of yet and planned for the future. Examples of those are are jumbo frames (frames with a payload bigger than 1500 bytes) and user-logic interrupt signals. The shell supports multiple target boards, in particular several of the AMD-Xilinx Alveo boards with UltraScale+ FPGAs. Its design is capable of handling up to four PCIe physical functions and two 100 Gbps ethernet ports. A key feature of the shell is the possibility of integrating custom logic blocks, that can manipulate network traffic. Specifically, it is possible to integrate custom logic with two different boxes that offer slightly different interactions with the data being transferred. One of the two boxes runs at 322 Mhz and the other at 250 MHz. Both have access to the data path, although they interact with different componets of the board, and consequently are allowed to alter packets as those pass through the NIC. Also, both have access to the system configuration. About the driver, it has support for multiple physical functions and multiple TX/RX queues for each PF. An important plus with regards to the other projects analysed in this section is that it benefits from AMD's support, also because it uses Xilinx QDMA IP to interface with the host through PCIe. In this research project, the focus will be on the shell and on the Linux kernel driver.

2.3 Corundum

Corundum [3] is a completely open source project that provides an FPGA-based NIC for in-network computing in System Verilog. This solution is capable of handling line rates up to 100 Gbps. Corundum offers the possibility of mapping multiple physical ports to the same interface, this is useful to aggregate multiple links and increase the bandwidth (shown in Figure 2). Each port has a different transmit scheduler with configurable parameters. Each interface is exposed to the OS and has its own separate transmit and receive queue: so by changing the scheduler settings

it is possibile to prioritize traffic and implement different queue management policies, without the need of changing the code.



(a) Traditional NIC, port assignment made in software

(b) Corundum NIC, port assignment made in hardware

FIGURE 2. Port assignment comparison

The design is fully modular, this means that it is possible to co-optimize the software and the hardware. A key difference with regards to OpenNIC is that Corundum uses a fully custom DMA subsystem: host to NIC communication uses descriptor queues and NIC to host communication is handled via completion queues. Both types of queues reside in DMA-accessible system memory. A custom segmented memory interface is also implemented: it allows to achieve high performace DMA over PCIe. The authors in the paper explain that they've used both the memory-mapping interface and the streaming one when designing Corundum. They also provided an extensive open-source simulation framework and a driver which can operate directly with the standard Linux network stack. Corundum also provides an application section for implementing custom logic. The application section has a number of interfaces that provide access to the core datapath and DMA infrastructure. A key difference with Ensō is that Corundum does not offer a C++ library to interact directly with the NIC and is not built around the concept of Ensō pipes.

2.4 sPIN and PsPIN

sPIN [4] is a programming model for in-NIC compute: it allows users to define handlers, simple processing tasks, that are computed on the NIC exploting hardware acceleration reducing the load on the host. This model is able to enable CUDA/OpenCL-like accelerators, providing user-level interaction. Applications can define handlers, those are then cross-compiled to run on the NIC. An additional capability specified in the model is that sPIN is able to match packets with messages.
The authors of the sPIN paper state that at the moment there isn't a SmartNIC design which allows to fully implement the sPIN model with all its features, so they propose their own implementation. That is PsPIN [4], a design implementing sPIN. It has a modular architecture with clusters of processing elements (HPUs), each HPU is a RISC-V core. Each cluster has its own L1 cache, cross-accessible, and the design features a shared L2 cache. An interesing capability of PsPIN, very useful when designing processing tasks that must work at high line rates, is the ability to detect handlers that cannot process at line-rate. PsPIN is not able to handle such situations on its own, it just raises an exception that helps programmers understand where the bottleneck is.

2.5 PANIC

PANIC [5] is a SmartNIC design aimed at giving cloud providers the ability to offer their clients hardware acceleration for networking tasks on a shared board. In such a scenario, it is key to guarantee isolation and fair scheduling of the available resources on the board and PANIC aims at providing exactly that.

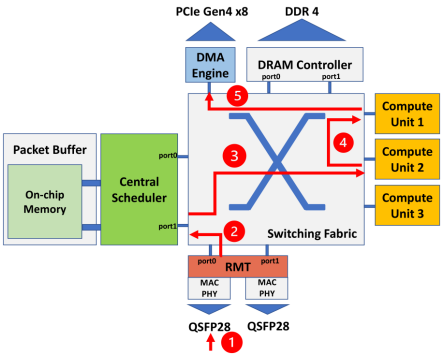


FIGURE 3. PANIC architecture

The figure 3 shows the design of PANIC, which is made of 4 components: an RMT switching pipeline [6], a switching fabric, central scheduler and multiple self-contained compute units. Following the numbers in the figure one can understand the steps a packets follows when entering the system. The RMT pipeline handles pre-processing of the packets (1 and 2): among other operations, it appends metadata specifing which compute units each packet has to be processed by (this information is stored in a PANIC descriptor). The central scheduler (3 and 4), in addition to performing scheduling operations, has access to buffers used to store packets when the next compute unit the packet has to go to is busy. Coupled with the possibility of replicating compute units, this allows PANIC to support below-line-rate compute units without compromising the throughput of the system. One of the features provided by this design is load-aware packet steering to allow parallel computation. Finally, packets are delivered to the host (5) through the DMA engine.

2.6 Pigasus

Pigasus [7] is a design implementing an intrusion prevention system able to achieve 100 Gbps on a single server. The key insights for the research behind it was to support packet analysis at 100 Gbps. Basically, packet analysis means matching packets against a set of signatures: this operation has to be performed on the payload, too and not only on the header. To understand whether a packet can be allowed to pass, Pigasus must also reassemble the TCP stream and this operation, in conjuction with the regex matching for

the signature, constitutes a resource intensive task. To achieve the desired performances, such a design must make a very efficient use of BRAM, basically only this kind of memory is used due to its low access times.
There are two interesting strategies employed by the team behind the Pigasus paper: the first one is about the regex matching challenge. Under the hypothesis that the majority of the traffic is not malicious and therefore would not match any signature, the regex matching operation is performed only partially on the FPGA. The computations done on the board are aimed at allowing as soon as possible those packets that are not a match, if a packet matches up to a certain percentage a signature (e.g. 95%), than it is passed to the CPU for the complete matching process. This ensures that only traffic with a high probability of being blocked will actually be slowed down. The second interesting design choice is about the TCP reassembly operation: there is a fast path for in-order reassembly (where the FPGA expects the sequence number to increment by 1 for each packet) and a slow path for out of order traffic. The slow path can handle a lower throughput, because it has to store packets in a buffer until the missing one arrives, but this is fine since most of the traffic is in order.

2.7 ACCL+

The research paper ACCL+ [8] aims at providing a collective engine for distributed applications. As nowadays multiple FPGAs are used in datacenters to accelerate applications, there is the need for an engine allowing boards to exchange data. As applications rely on complex communication patterns, one of the drivers for ACCL+ is to provide a collective communication resource that does not need to rely on CPU operations. This project offers portability across communication models, working with the Message Passing Interface (MPI) and the streaming ones. It supports applications that may execute partially on the CPU and partially on the FPGA. ACCL+ integrates with the AMD's developing environment Vitis, supporting UDP and TCP offloading.

2.8 SoA conclusions

The SmartNIC projects analysed in this section are all aimed at providing high performance network interfaces, each meant for a specific use case. The main differences between them are the features they offer and the target applications. As the focus of this research project is to port Ensō onto an AMD/Xilinx board, this is the main driver for the choice of the candidate best suited for hosting the porting. Even if all the projects offer interesting design choices and features, the only ones that easily allow to integrate custom logic for generic traffic manipulation are OpenNIC and Corundum. From a technical point of view, they differ in the way they handle the DMA subsystem and the memory interface. The choice of OpenNIC was made because it uses the Xilinx QDMA IP core for the host interface, while Corundum uses a fully custom DMA system. As a result, OpenNIC benefits from mainstream support for the QDMA IP and should also benefit for AMD's support for the rest of the code. This is the main reason why OpenNIC was chosen, so that by having a more supported environment it would be easier to port Ensō and the final result would benefit from more compatibility. This step also allowed to research different approaches for dealing with the porting, as it will be explained in the next section.

3 Methodology and challenges

3.1 Where to implement the Ensō logic

The main result of the state of the art phase was having found a project, running on AMD/Xilinx boards, than can potentially be suitable for hosting Ensō. At this point, it was needed to focus on the functionalities offered by OpenNIC useful for the porting and finding out how to start integrate the Ensō logic. OpenNIC offers two possibilities to integrate a custom logic block into its design: one running at 322 MHz and another one running at 250 MHz. The figure 4 shows the design of the shell and is taken from the OpenNIC repository.

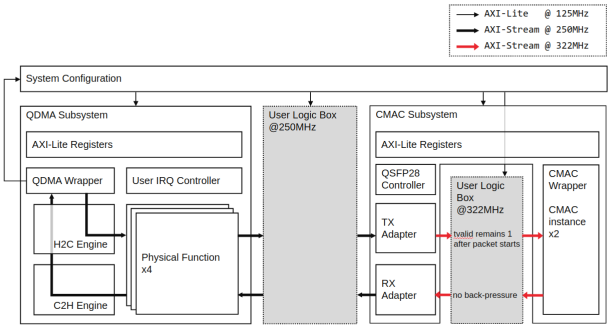


FIGURE 4. OpenNIC Shell design

Both feature two different kind of interfaces: an AXI4-stream interface to manipulate packets being handled by the board and an AXI-lite interface to access dedicated registers and interact with the configuration of the system. It is key that these custom logic blocks have full access to packets, otherwise they would not be suitable for running the logic of Ensō. By looking at the design of the shell, shown in Figure 4, it is possibile to see that the two blocks differenciate for the position in which they are placed with regards to the data path of the board. The CMAC kernel is an RTL kernel which encapsulates the UltraScale+ Integrated 100G Ethernet Subsystem, interfacing the ethernet ports with the rest of the board. The 322 MHz block interfaces directly with the CMAC ports as it is part of the CMAC subsystem: the documentation in the repository states it is intended for computations that need access to packets as soon as they are picked

up from the wire or are just ready to be transmitted on ethernet.
Indeed, the logic for this block is supposed to run at a higher frequency to keep up with the data path. Also, as this block is supposed to process data as soon as it is received, the AXI interface on the receiving side does not have the TREADY signal: the block cannot make the bitstream on the wire wait. TREADY is used by the receiving entity in the AXI communication to signal to the sending one it is possible to send data. As the 322 MHz block is connected to ethernet ports, it makes no sense to expect packets start and stop arriving respecting the meaning of a TREADY signal. Consequently, the design of the block must allow it to always be ready to receive data. On the other hand, the 250 MHz block uses an AXI4 interface without this kind of restriction: it communicates with RX/TX adapters and with engines that then exchange data with the PCIe side of the NIC. For the scope of the porting, the higher frequency of the 322 one is not need and nor is the direct communication with the CMAC ports, so it is better to opt for the 250 MHz block to put the logic into. From the state of the art up to this point, the methodology was clear and simple: identify the features offered by each alternative and opt for the one with more benefits being the nearest to the need of the research project. It was then needed to start dissecting the details of Ensō pipes.

3.2 Ensō pipes

At first, it was not clear whether Ensō implemented some kind of dynamic reconfiguration. That could have been the case, as the paper advertises a high adaptability to different applications and does not mention a cap to the number of simultaneous pipes supported. However, an in-depth analysis of the source code revealed this is not how the library manages buffers: when the application requires for a new Ensō pipe, the library just designates a previously defined portion of memory for that specific circular buffer. This also reveled that the operation of starting a new communication link requires interaction with the board. This is done exploiting a tight coupling between the C++ library and the HDL code. In these interactions, it appears that the kernel driver communicating with the FPGA does not play any role, other than allowing the library to access configuration addresses on the FPGA. Indeed, the driver used by Ensō is the standard one provided by Intel for its Stratix boards.
Therefore, it was clear that even reusing as much as possible of the C++ library it was needed to adapt and rewrite all the C++ code interacting with the NIC, throught the driver, using vendor specific primitives. In addition to that, the interface Ensō uses to communicate with the host is radically different with regards to a Network Interface Controller: typically it is up to the OS kernel to manage queues and buffers given to processes, while in this scenario the C++ library offers these functionalities heavily relying on the hardware design. Instead, the driver provided by OpenNIC is one that makes the OS communicate with the card as if it were a standard NIC. The OpenNIC code is designed in such a way that the operating system does not have to deal with the fact the network is managed by a network card and an FPGA, but it just treats the device as a standard one.

3.3 Interaction with the driver

Looking at figure 5 [9] helps understand how function calls from applications are translated into hardware interactions by the kernel. With OpenNIC, and any standard NIC, the user apps are actually code written by developers building an application, whilst in the Ensō scenario the C++ library plays the role of the user app after having elaborated invocations originally made by callers.

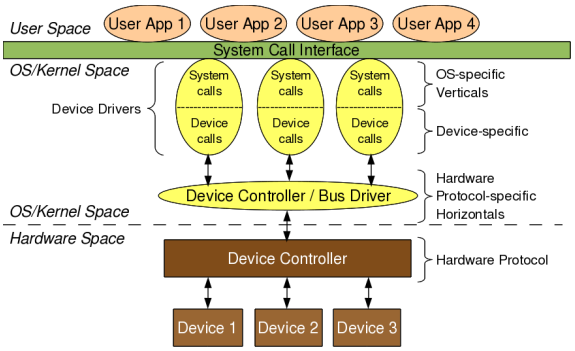


FIGURE 5. Linux device driver

The driver written by Xilinx for OpenNIC is used by Linux to have the board perform network operations. This is a key difference: Ensō's driver is just allowing the C++ library to interact with the FPGA. Having found out this meant it was necessary to work parallely on the C++ library and the HDL code. This definitely constitutes a major challenge, but it is not the only one. On top of the offloading capabilities, Ensō radically changes how data from and to the application is delivered to and from the card. The shell of OpenNIC, as well as any other project analysed in the state of the art, allows to manipulate packets in the data path, but it does not offer a way to interact with the application. It is possible to have the custom logic block interact with the system configuration. Potentially, such communication could be exploited to have the driver offer some kind of interfacing functionalities with such logic block, but this would mean completely abandoning the rest of the OpenNIC driver managing packets. Another way of presenting this incompatibility is the following: the first founding block of the Ensō paper is to change the fixed-size packetized abstraction and the whole OpenNIC shell was designed strarting exactly from this abstraction. The research of this project revealed that a possibile way to achieve Ensō pipes abstraction is to rewrite the

driver of OpenNIC to change the packets handling logic and to interact directly with the custom block through the AXI-lite interface. Of course, these changes should be reflected onto the shell's code, too. This analysis clearly allows to underline the differences between Ensō and OpenNIC that do not allow for a complete integration without a substancial rewrite of source code. However, it is still possible to target some of the additional features offered by Ensō and work on porting those. For the reasons explained here, in conjunction with the limited scope of this research project, it was decided to develop a demo to show the feasibility of the integration of some of Ensō offloading capabilities with OpenNIC, without trying to implement the whole logic.

3.4 MAC filtering

One of the first operations Ensō must perform, in order to offer networking capabilities, is to check whether the destination MAC address on each packet matches the one of the card. The porting of this mechanism is key to understand how to port Ensō. Although it being simple, it is very explicative of the methodology and the process needed for the porting to be successful. Also, being able to perform this verification means having successfully integrated some custom code with the shell of OpenNIC. This MAC filtering is the logic chosen to be shown as demo in this resource project to prove the feasibility of the task. The source code of the shell is written in Verilog, one of the innovations brought with this project is having developed the demo code in High Level Syntesys code. This integration between two diffrent languages, working at a different level, was one of the challenges. From a methodology point of view, this means being able to elevate the abstraction level at which the code is written, making it easier to understand and to modify. To do so, the demo IP is developed using Vitis HLS and then the generated Verilog code is integrated with the shell of OpenNIC.
Luckyly, the custom logic box in the OpenNIC shell is required to communicate using the AXI4-stream interface, which is supported by Vitis HLS using the hls_stream and the ap_axis libraries. To make sure the signals generated by Vitis match exactly the ones expected by the shell, the latest version of Vitis, 2023.2, is used: it supports customizing the AXI stream chosing which signals to include in the interface.

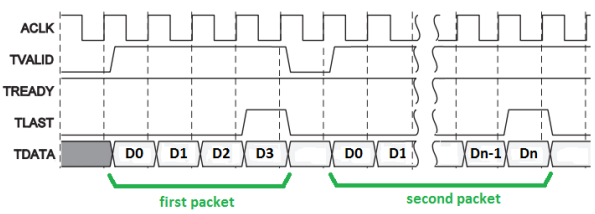


FIGURE 6. AXI stream interface

Figure 6 shows a simple example of a transmission with AXI and the base signals that must be included in the AXI stream interface. In addition to those, signals TUSER, TID and TDEST in HLS are enabled to be mapped onto tuser_user, tuser_src and tuser_dst of OpenNIC, as per documentation.
The logic of such module is simple: the MAC address is read from the header of the packets, in the first batch of the AXI stream communication, then the packets is let through only if such MAC matches the one of the board. At the moment, the MAC address is set at compilation time in Vitis HLS; this is not a major limitation as the MAC address of a board never changes. Being able to read such address at runtime is left for future work. The generated Verilog code is then integrated with the shell, via a simple wrapper, and the whole project is synthetized using Vivado.

3.5 Building and testing

The testing of the correctness of the MAC filter is done using the Vitis HLS environment. A testbench is used to call the HLS function and Vitis cosimulation feature guarantees, at least in the simulation environment, that the behaviour of the synthetized design corresponds to the one of the HLS code. To ensure the testbench is as effective as possible, it uses real data captured with Wireshark. The synthetized design of OpenNIC with and without the custom module can then be loaded onto the FPGA board. This step was definitely challenging, as the documentation is not very detailed. The .tcl script provided with the source code for the FPGA programming needed to be adjusted for it to properly work, as it missed some fundamentals configuration parameters and was not retrieving those from the configuration files properly.
After this step, there was the need to physically test the achieved results. Due to the lack of compatible systems and of a developing infrastructure allowing to test SmartNICs, the only available alternative was to activate the loopback feature of the board. That enables an hardware functionality looping back to the host each packet that is meant to be sent on the network. Using such feature would allow to properly verify whether packets were being filtered correctly and quantify the potential performance impact of the custom logic. This operation should have been simple, but even after extensive testing and experimenting it was not possible to have the board activate such functionality. Leastwise, the programming phase completed successfully, the board was correctly detected using the proper driver and the link was treated as UP by the operating system. When able to measure packets being processed, the plan to fully test this project would be to use a packets generator and measure the throughput of the board with and without the custom logic. This would allow to quantify the performance impact of the custom logic, in addition to making sure only packets with a matching MAC destination address are let through. At the moment of writing this is left for future work.

4 Experimental achievements

This research project, through the state of the art analysis, identified two valid candidates to host Ensō: OpenNIC and Corundum. Also, the key features needed for the porting to be successful were clearly underlined. It is now evident that, without a partial rewrite of both the C++ library of Ensō and the HDL code defining the design of the target board, it is not possibile to fully port Ensō onto an existing SmartNIC. Also, the driver of the target SmartNIC of the porting, OpenNIC, needs to be modified to support the new logic. The studies within this project allowed to define a methodology to follow to achieve the porting. That is start from a more detailed analysis of the C++ library and adapt the driver of OpenNIC step by step to allow the interactions needed by the library.
Thanks to the MAC filter, it was shown that some of the features offered by Ensō can be integrated with OpenNIC. Additionaly, the successful build of the project and the FPGA programming shows that the custom logic block can be integrated with the shell and that the board supports the custom block. That is true even if the loopback feature could not be activated. However, due to the lack of an infrastructure allowing to test SmartNICs, it was not possible to fully test the achieved results. As the MAC filter was developed using Vitis HLS, it shows it is possible to elevate the abstraction level at which the code is written. To this extent, it was proved it is possibile to properly specify an AXI4-stream interface in HLS so that the generated System Verilog code is compatible with OpenNIC.

5 Conclusions and future work

This report presents the work done with the SmartNIC research project in the context of the High Performance Processors and Systems course. The goal was to analyse the feasibility of porting the Ensō design [2] onto an AMD board. This explorative task starts from conducting a state of the art study to find, among available solutions developed by the research community, the best candidate to host Ensō. OpenNIC was chosen as the best one, as it allows to integrate custom logic blocks and should benefit from AMD's support. Ensō pipes are one of the main features offered by Ensō, completely integrating the logic of such mechanism revealed to be unfleasible in the available time frame. A demo, a MAC filter, was developed to show the feasibility of integrating some of the offloading capabilities of Ensō with OpenNIC. The custom block was developed using Vitis HLS and then integrated with the OpenNIC shell. FPGA programming and NIC detection by the OS were completed successfully, after the custom logic was checked using testbenches.

5.1 Future work

The next steps would be to fully test the project, using a packets generator to measure the throughput of the board with and without the custom logic. This would allow to quantify the performance impact of the custom block and to make sure only packets with a matching MAC destination address are let through. In addition to that, the C++ library of Ensō should be adapted to interact with the OpenNIC shell: the buffer allocation should be changed to mimic the logic of Ensō pipes and the driver should interact directly with the custom logic block. This would require a partial rewrite of the library, as the current one communicates using Intel's proprietary driver; and for sure modifications to the shell code would be required, too. The work done in this research project is a first step towards the integration of Ensō with OpenNIC.

References

[1] Daniel Firestone; Andrew Putnam; Sambhrama Mundkur; Derek Chiou; Alireza Dabagh; Mike Andrewartha; Hari Angepat; Vivek Bhanu; Adrian Caulfield; Eric Chung; Harish Kumar Chandrappa; Somesh Chaturmohta; Matt Humphrey; Jack Lavier; Norman Lam; Fengfen Liu; Kalin Ovtcharov; Jitu Padhye; Gautham Popuri; Shachar Raindel; Tejas Sapre; Mark Shaw; Gabriel Silva; Madhan Sivakumar; Nisheeth Srivastava; Anshuman Verma; Qasim Zuhair; Deepak Bansal; Doug Burger; Kushagra Vaid; David A. Maltz; Albert Greenberg. "Azure Accelerated Networking: SmartNICs in the Public Cloud". In: (2018). URL: https://www.microsoft.com/en-us/research/uploads/prod/2018/03/Azure_SmartNIC_NSDI_2018.pdf.

[2] Hugo Sadok et al. "Enso: A Streaming Interface for NIC-Application Communication". In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, July 2023, pp. 1005–1025. ISBN: 978-1-939133-34-2. URL: https://www.usenix.org/conference/osdi23/presentation/sadok.

[3] Alex Forencich; Alex C. Snoeren; George Porter; George Papen. "Corundum: An Open-Source 100-Gbps NIC". In: (). DOI: 10.1109/FCCM48280.2020.00015.

[4] Salvatore Di Girolamo; Andreas Kurth; Alexandru Calotoiu; Thomas Benz; Timo Schneider; Jakub Beránek; Luca Benini; Torsten Hoefler. "A RISC-V in-network accelerator for flexible high-performance low-power packet processing". In: (). DOI: 10.1109/ISCA52012.2021.00079.

[5] Jiaxin Lin et al. "PANIC: A High-Performance Programmable NIC for Multi-tenant Networks". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).*

USENIX Association, Nov. 2020, pp. 243–259. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/lin.

[6] Pat Bosshart; Glen Gibb; Hun-Seok Kim; George Varghese; Nick McKeown; Martin Izzard; Fernando Mujica; Mark Horowitz. "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN". In: (2013). URL: http://yuba.stanford.edu/~grg/docs/sdn-chip-sigcomm-2013.pdf.

[7] Zhipeng Zhao et al. "Achieving 100Gbps Intrusion Prevention on a Single Server". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1083–1100. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng.

[8] Zhenhao He; Dario Korolija; Yu Zhu; Benjamin Ramhorst; Tristan Laan; Lucian Petrica; Michaela Blott; Gustavo Alonso. "ACCL+: an FPGA-Based Collective Engine for Distributed Applications". In: (). URL: https://arxiv.org/html/2312.11742v1.

[9] "Linux Drivers". In: (). URL: https://sysplay.github.io/books/LinuxDrivers/book/index.html.