


Full-System Evaluation of the sPIN In-Network-Compute Architecture

Master Thesis

Author(s):

Xu, Pengcheng 

Publication date:

2023-09

Permanent link:

<https://doi.org/10.3929/ethz-b-000637676>

Rights / license:

In Copyright - Non-Commercial Use Permitted



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Full-System Evaluation of the sPIN In-Network-Compute Architecture

Master Thesis

Pengcheng Xu

September, 2023

Advisors: Prof. Dr. Torsten Hoefler, Mikhail Khalilov, Timo Schneider

Department of Computer Science, ETH Zürich

Abstract

In-network-computing with SmartNICs is gaining popularity in high-performance networking for their ability to offload packet processing tasks from the CPU and their latency advantage thanks to the proximity to the network traffic without having to go through PCIe. The sPIN in-network-computing paradigm developed at ETH Zürich aims to provide a programming model for developers to build high-performance packet processing routines for on-path SmartNICs. While the paradigm has been evaluated with use cases from diverse scenarios in software and hardware simulation, it has yet to see a full E2E system-level evaluation that exercises the entire packet processing loop on hardware in the real world. In this thesis, we perform an end-to-end analysis of the sPIN paradigm by building a full-system prototype of sPIN on FPGA based on PsPIN, a cycle-accurate simulation prototype of sPIN, and Corundum, an open-source FPGA-based Ethernet NIC. We show that the resulting system FPsPIN facilitates the development and testing of sPIN handlers, allowing real-world performance and computation/communication overlap evaluations that would not have been possible with the old cycle-accurate simulation models due to the slow simulation speed and absence of a host CPU. We present various improvement suggestions to the sPIN specification, discovered through the process of building FPsPIN. In addition, we conduct a detailed performance evaluation of FPsPIN through three benchmarks implemented for the platform, showing a 50 us latency advantage, over 99% computation/communication overlap, 6.4 Gbps and 1.2 Gbps throughput in simple and complex synthetic benchmarks. The lower application throughput shows the deficiency of the packet processing cores used in FPsPIN and shows an opportunity for future research on desirable architectural features for SmartNIC cores.

Acknowledgements

I would like to express my most sincere gratitude to the advisors of this thesis, Mikhail Khalilov, Timo Schneider, and Prof. Dr. Torsten Hoefler, for their valuable feedback, guidance, and especially support when it is most needed, throughout the entire thesis project. I believe that the professionalism, passion, and patience I experienced working with them will have a deep, long-lasting positive impact on my future career.

I would like to heartily thank the members of the NetOS Group at ETH Zürich, for their invaluable support in diverse forms. These include but are not limited to punctual invitations to lunch and Super Kondi [1], unlimited witty jokes, medal-winning master thesis examples, brilliant talks about old computers, and much more. I owe my successful completion of this thesis to the acquired mental strength from the time spent with them.

I would also like to thank Junchi Chen and Zhehan Fu, for the many evenings I have spent with them of experimental cooking, drinking, and complaining about various issues in life. Their sincereness and accompany have been indispensable in helping me manage stress and maintain mental health throughout the project and beyond.

I would like to thank my family, Tingjin Xu and Yaping Guo, for their unlimited and unconditional love ever since I was born and for the decades yet to come. My life journey and academic endeavours would have not been possible without their support all the way.

Last but not least, I would like to thank *you*, my dear reader, for spending your precious time reading this thesis. It is with your attention and support that this work could have its influence in the future.

Contents

Contents	iii
1 Introduction	1
1.1 Contributions	2
2 Background	5
2.1 SmartNIC Architectures	5
2.2 sPIN	7
2.3 FPGA and Design Reuse	9
2.4 PsPIN and RISC-V	11
2.5 Corundum	12
3 Hardware	15
3.1 Control Path	16
3.2 Data Path	20
3.2.1 Ingress	20
3.2.2 Egress	24
3.3 Host DMA	24
3.4 Design Considerations	25
4 Software	27
4.1 CPU Kernel Modules	27
4.1.1 mqnic.ko	28
4.1.2 mqnic_app_pspin.ko	29
4.2 CPU User-Space	31
4.3 Handler Runtime	34
5 sPIN Revisited	37
5.1 Messaging and Reliability Layer: SLMP	37
5.1.1 Motivation	37
5.1.2 The solution	38

5.1.3	SLMP flow control	40
5.2	Telemetry	41
5.3	Scheduler Concurrency Control	42
5.4	Network-layer Protocol Handling	42
5.5	Handler Initialisation	43
5.6	Host-side Activation	44
5.7	Alternative Host DMA Interface	44
6	Evaluation	45
6.1	Experiment Setup	45
6.2	Design Analysis	46
6.3	Demo Applications on Real Hardware	48
6.3.1	Ping-pong	48
6.3.2	MPI datatypes	52
6.3.3	SLMP file transfer	57
7	Future Work	61
7.1	Improving F_{\max}	61
7.2	Architectural Exploration for HPUs	62
7.3	Advanced Flow Control in SLMP	62
7.4	Parallelise Packet DMA and Scheduling	63
7.5	Host Notification Queue Pair	63
7.6	More Real-world Applications	64
7.7	Explore Functionalities from Corundum	64
7.8	Stability & Bug Fixes	64
8	Conclusion	65
A	Reproducing the Results	67
A.1	Building the System	67
A.2	Running the Experiments	69
B	Debug Facilities	71
B.1	Hardware	71
B.2	Software	72
B.3	Quirks & Workarounds	72
	Bibliography	75
	Acronyms	83

Introduction

The network is the computer.

– John Gage, *slogan for Sun Microsystems*

Fourty years since John Gage has coined the catchy slogan for Sun Microsystems, networked computers are being used around the world more than ever, empowering a vast range of modern technologies that our societies are increasingly depending upon. This directly corresponds to the ever increasing higher bandwidth of datacenter networks: 100 and 200 Gbps Ethernet and InfiniBand are becoming de-facto standards; link speeds of 400 Gbps and beyond are already being developed and deployed [2].

On the other hand, CPU processing power have not been scaling up at the same pace with the increase of link speeds and the higher packet processing overhead that comes with these speeds. *remote direct memory access* (RDMA) is one of the many offloading technologies designed to reduce packet processing overheads on the CPU. While the RDMA approach reduces network-related processing overheads, the actual consumption of the packet payload still happens on the CPU cores. Modern CPU cores are built with complicated micro-architectures optimised for compute-heavy workloads instead of most packet-processing workloads that involve simple arithmetics and data movement. The CPU cycles spent processing packets would be best utilised to perform the compute-heavy tasks instead, which is what the CPU is designed for.

SmartNICs are a recent movement towards offloaded packet processing to free the CPU from packet processing and thus spend more time handling the typical computation tasks. They come in different programming models and dataflow models. Among different paradigms, sPIN [3] developed at ETH Zürich proposes network accelerators with a micro-architecture optimised for packet processing and fine-grain memory hierarchies and data movement

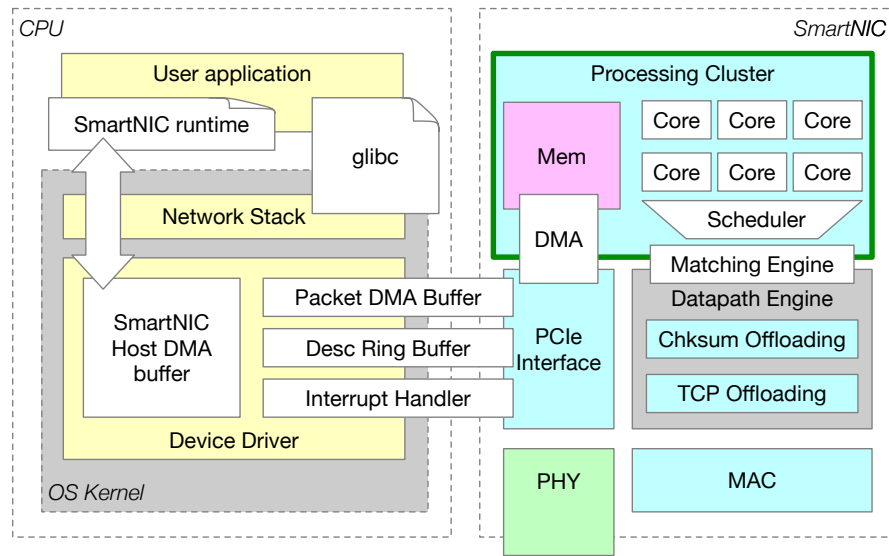


Figure 1.1: Overview of the complete server system, showing the software stack on the CPU and hardware components on the SmartNIC. The green box marks the existing PsPIN processing cluster available from previous work. Everything else needs to be developed, integrated and tested.

acceleration. It offers precise control to the programmers to build high-performance networked applications that are offloaded completely to the SmartNIC.

The sPIN paradigm has been evaluated extensively with diverse networked applications [4, 5, 6], showcasing its capability of offloading complicated applications to a sPIN-based network accelerator. Up to the writing of this thesis, however, all evaluations of sPIN took place in simulation and there lacked a real-world end-to-end demo on hardware. While simulation works well to demonstrate capabilities of the paradigm in a *synthetic* environment, an end-to-end evaluation involving all parts of the final system would uncover unforeseen design and implementation shortcomings and offer valuable insights to further improve the paradigm.

1.1 Contributions

In this section of the introduction, we give a brief summary of the contributions of this thesis and references of where they are explained in the text. An overview functional diagram of the system is shown in Figure 1.1.

First of all, we built *FPsPIN*, the first full-system demo of sPIN in hardware based on the PsPIN [7] implementation of sPIN and the Corundum [8] open-source Ethernet NIC. This allows fast testing of packet *handlers* (code that runs on the sPIN cluster, more details in Section 2.2) in comparison to the

slow cycle-accurate simulator (Section 2.4). The hardware (Chapter 3) and software (Chapter 4) components bridge the missing parts in the PsPIN prototype to allow sending and receiving of packet data from real NICs and, most importantly, completes the *host-side* programming model of sPIN. This allows development of complete sPIN applications with both the NIC-side handlers and host-side application. In all, the demo system greatly facilitates development both of the sPIN platform as well as applications designed for it.

An important yet largely unexplored benefit of sPIN is the possibility of *computation/communication overlap* by offloading packet processing tasks to the SmartNIC. We implement synthetic ping-pong and file transfer benchmarks to demonstrate the E2E latency and throughput of the system in ideal situations. We port the MPI Datatypes [9] sPIN handlers [4] to the FPsPIN platform (Section 6.3.2) to demonstrate the ratio of overlapping between the computation and communication tasks, as well as interference from each other. These demonstrations show sPIN's potential of accelerating networked applications and improving efficiency, but also open up interesting research questions about the architectural design of packet processing units in SmartNICs.

Last but not least, we discovered numerous shortcomings and points of improvement in the sPIN specification [3] (Chapter 5) during the development of the FPsPIN prototype system. We discuss about the issues closely with the sPIN team and work together towards a more complete and sensible specification for other implementations of the paradigm. Several of the proposed changes have already been incorporated back into the specification.

Background

The job of science will never be done, it will just sink deeper and deeper into never-ending complexity.

– Isaac Asimov, *The Secrets of the Universe*

In this chapter, we give an overview of the related technologies and prior works that are important to this thesis. These projects and ideas are the building blocks of the FPsPIN prototype system.

2.1 SmartNIC Architectures

SmartNICs from different vendors tend to have different architectures, but they can be classified by the *datapath design* largely into two categories: *on-path* versus *off-path* [10, 11]. We show a brief overview of both paradigms in Figure 2.1. In addition to data path designs, there is also an increasing trend to use *reconfigurable hardware* to implement SmartNICs. We introduce the various paradigms and discuss how most commodity SmartNICs fit into these categories.

On-path SmartNICs *On-path* SmartNICs, also known as *bump-in-the-wire*, have the *processing element* (PE) sitting on the packet processing path for the ability to modify incoming and outgoing packets. Figure 2.1a demonstrates the flow of packet data on these SmartNICs: incoming packets get assigned to NIC PE and either gets processed on the NIC as offloaded traffic (①), or steered to the host (②). The NIC PE can further interact with the host CPU over the PCIe interface (③). Examples of on-path SmartNICs include the Marvell LiquidIO [12], Netronome Agilio CX [13], as well as research systems [14, 15].

The most important benefit in this design is that the SmartNIC PEs have very low latency access to packet data (①), allowing efficient NIC-only transactions

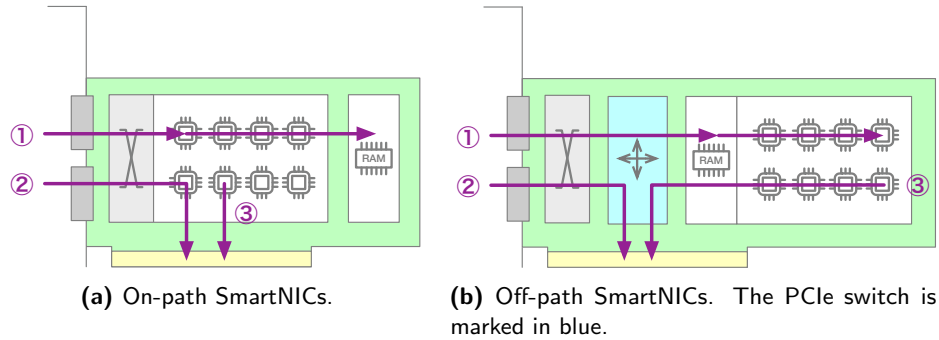


Figure 2.1: Overview of different PCIe-based SmartNIC data path designs. The purple arrows denote different data flow interactions between the NIC cores and the host CPU upon incoming traffic. ①: offloaded traffic; ②: (non-offloaded) host traffic; ③: host-NIC interactions.

(that do not involve the host). The downsides of this design mainly come from the fact that regular, non-offloaded host traffic still requires SmartNIC PE to steer to the host (②). This results in degraded host traffic performance when the NIC is heavily loaded [10]. Another difficulty is due to the low-latency requirement from the on-path nature of the SmartNIC data path, forcing a low-level interface that is difficult to program against.

Off-path SmartNICs In contrary to on-path SmartNICs, *off-path* SmartNICs have the packet PE, usually as a separate *system-on-chip* (SoC) off the regular packet data path. We show the flow of packet data on these SmartNICs in Figure 2.1b: compared to the on-path paradigm, an extra PCIe switch allows both the host CPU and the packet PE to act as fully-capable *hosts* to the NIC. The switch steers the incoming packets from the network fabrics to the NIC PE (①) or the host (②), according to configurable rules. An example of this SmartNIC design is the NVIDIA BlueField [16].

Thanks to the packet processor not being on the critical path to the host, the off-path paradigm allows for more complicated software stacks on the SmartNIC. This allows for a full network stack and operating system, usually the Linux kernel, on the SmartNIC cores, as well as more user-friendly programming interfaces. However, the addition of the PCIe switch significantly increases the latency of NIC-offloaded tasks (①) and NIC-host interactions (③) compared to on-path designs [11].

Reconfigurable hardware While most commercial vendors implement the SmartNIC PE as fixed-purpose SoC with special accelerators, there are increasing attention into building FPGA into the SmartNIC. This results in architectures that are either SoC coupled with an FPGA or entirely FPGA-centric designs. Examples include in-house deployments inside Microsoft

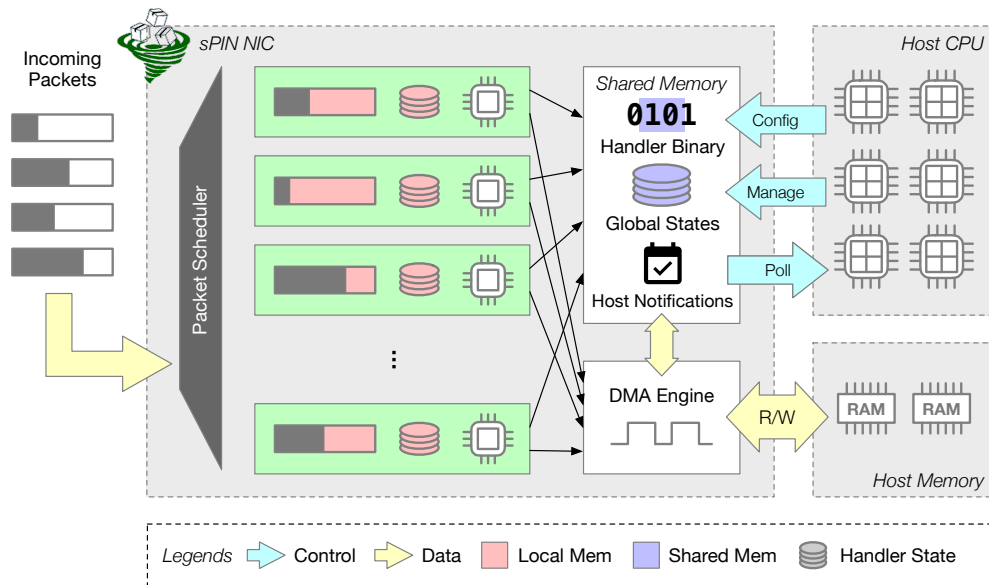


Figure 2.2: Overview of the sPIN architecture.

Azure [17], products from Intel [18] and AMD [19, 20], as well as research systems [15, 21].

FPGA-enabled SmartNICs excel in reconfigurability. While having purpose-built SoC coupled with a pre-determined set of accelerators can provide the highest possible performance, they cannot evolve to accommodate the changing demands of the workload. Reconfigurable solutions allow users to customise the SmartNIC hardware *after* the hardware is built by implementing new, custom accelerators in the FPGA. This offers the customers great flexibility and more possibilities for accelerated offloading. They are also suitable in domains with a small volume to the point where it is not cost-effective to tape out a custom *application-specific integrated circuit* (ASIC) with domain-specific accelerators, such as in telecommunications and research.

2.2 sPIN

sPIN [3] is a portable programming model that allows a programmer to offload simple packet processing functions of a networked application code the NIC. sPIN is designed to exploit packet-level parallelism through the execution of short, lightweight *packet handlers*. The *streaming* semantics of sPIN comes from its *flow-oriented* nature in keeping a state for each traffic flow¹ (e.g. TCP/UDP connections, MPI messages, etc.). This allows for efficient packet processing with higher expressiveness in comparison to other

¹We use *flow* and *message* interchangeably in the context of sPIN in this thesis.

2. BACKGROUND

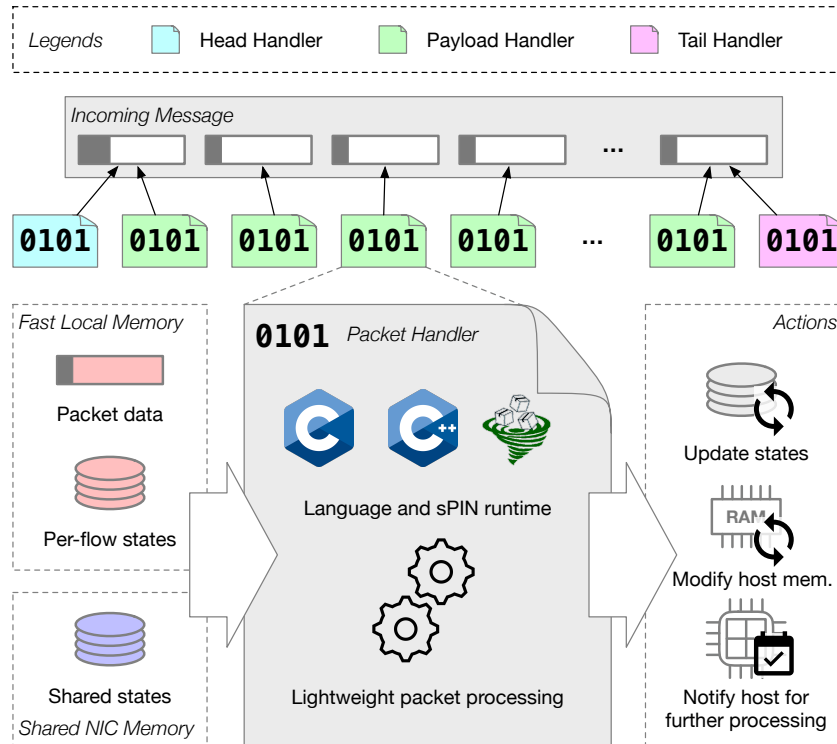


Figure 2.3: Consumption of packets of an incoming message by packet handlers. Inputs and actions of individual handlers are shown in the zoom-in view.

offloaded packet processing models, such as Portals 4 [22] or eBPF/XDP [23], in which packet processing functions are stateless. An overview of the sPIN architecture is shown in Figure 2.2.

Packet handlers and how they behave is the core of sPIN: the possible actions from the handlers together with the packet scheduling requirements define sPIN as a *network instruction set architecture* (NISA) [3]. We visualise how sPIN handlers process packet data in Figure 2.3. Handlers take the packet data and per-flow state as input and perform one or many of the following actions: *update the flow state*, *read from and write to host memory*, and *notify the host for further action*. On incoming packets of a flow, the programming model further defines three types of handlers for execution on the packet PE:

- A *head* handler to be executed on the first packet of a flow; this is guaranteed to be the first handler executed for a given flow.
- A *payload* handler to be executed on every packet; they are guaranteed to be scheduled only after the *head* handler finishes execution.
- A *tail* handler to be executed on the last packet of the flow; this is guaranteed to be scheduled only after all *payload* handlers finish execution.

Packet handlers are scheduled and executed concurrently on the SmartNIC PE, namely *handler processing unit* (HPU). A *packet scheduler* schedules incoming packets on the HPUs for execution w.r.t. the handler scheduling dependency requirements. The HPUs access packet data and per-flow states in fast local memory and shared states in the shared NIC memory. They access host memory through a device-level *direct memory access* (DMA) engine between the host and shared NIC memory. The host CPU functions as the control plane and is in charge of programming the scheduler and the HPUs, host and NIC memory allocation and initialisation, as well as processing host notifications from the HPUs.

2.3 FPGA and Design Reuse

FPGA [24] is a type of integrated circuit that allows runtime *reconfiguration* after being manufactured. They consist of an array of *configurable logic block* (CLB) that can function either as *look-up table* (LUT) or *flip-flop* (FF), on-chip SRAM as *block RAM* (BRAM) or *Ultra RAM* (URAM) macros, and programmable routing resources that connect the input and output of CLBs. LUTs are used to implement combinational logic and FFs for sequential logic. FPGA also have high-speed transceivers implemented as hard macros for high-speed buses, e.g. PCIe or DDR4. A *bitstream* for an FPGA, when flashed onto the device, configures all CLBs and routing resources into a specific digital logic design. FPGA can be used to validate hardware designs before they are taped out into ASIC since large designs take way too long to simulate and bring up complicated software. They are also used for situations that call for reconfigurability, such as building custom accelerators in the cloud, as well as for products with a small volume where producing ASIC is not cost-efficient.

Static timing analysis (STA) As is the same with traditional digital design on ASIC platforms, implementing such on FPGA requires STA such that sequential logic in the design can capture signals synchronously and consistently. For data paths between clocked elements (FF), the *electronic design automation* (EDA) tool calculates the time it takes from the signal to propagate from the source to the destination, such that the signal is captured in the same clock cycle. The time difference between the required and actual signal arrival time through the wires and combinatorial logic (LUT) is called a *setup slack*; a *negative* setup slack indicates that the signal arrived too late i.e. *missed* the clock edge at the destination. *Hold slack* is defined similarly for the requirement that the signal value should be held stable until the destination has captured the signal. Commercial EDA tools use the metrics *worst negative slack* (WNS) and *worst hold slack* (WHS) to evaluate if a design has passed

timing, and *total negative slack* (TNS) and *total hold slack* (THS) for how badly the design failed to close timing.

Since large FPGA designs may take hours to implement, it is important to identify the root cause of a timing violation for resolution. The design may contain overly long combinatorial paths that exceed the clock period budget; this requires either a redesign to split these paths into multiple clock cycles (i.e. *retiming*), or a lower clock frequency (F_{\max}). On the other hand, local exhaustion of routing resources would cause *routing congestion*, forcing the EDA tool to take long detours when routing signals. Congestion issues require the designer to provide assistance in placement by drawing *placement block* (PBlock), also known as *floor-planning*, or to reduce the overall resource consumption. Last but not least, since STA is a pessimistic analysis based on the worst characteristics of the device (e.g. temperature, power, etc.), a small timing violation may still result in a functioning design under normal operating conditions.

Design reuse To facilitate the development of new microchips, hardware *intellectual property* (IP) vendors package their hardware function blocks as *IP blocks* and redistribute them to customers for integration into their design. For interoperability with other IP vendors as well as customer designs, IP blocks adopt industrial standards for *bus protocols*. On the other hand, if design components do not speak the same bus protocol, an *adapter* is needed. Adapters consume hardware resources and may impact performance depending on the complexity of the protocol; a perfect adapter may not even be possible in case of a mismatch of semantics between the protocols.

The simplest bus protocol for signalling the validity and acknowledgement of data is the *ready-valid* protocol. It consists of two extra wires in addition to the data signals: *ready* denotes that the receiver can accept data, and *valid* denotes that the data-driven by the sender is valid. A *beat* of data is successfully transmitted if both ready and valid are held high for one clock cycle; the receiver can assert *back pressure* by de-asserting the ready signal. A common variant of the protocol is a *valid* protocol, where the ready signal is missing and implied to be always high. This indicates that the sender has no internal buffer for dealing with possible back pressure from the receiver and that the receiver always has to be ready for data. If the receiver could not keep up with the data rate on the bus, a valid beat of data would be dropped.

The *Advanced eXtensible Interface* (AXI) [25, 26] is a family of *on-chip* communication bus protocols designed to connect IP blocks in a hardware design. AXI protocols follow a *master-slave* design where the bus master initiates transactions and the bus slave responds. The protocol has three flavours, designed for different use cases. *AXI-MM* is designed for high-performance memory-mapped read and write access from processor-like masters on addressable

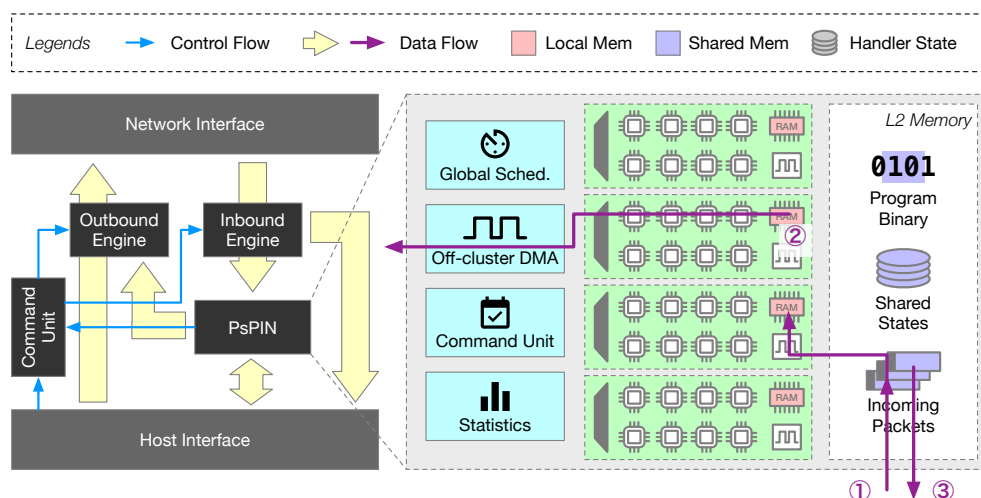


Figure 2.4: Overview of the PsPIN architecture and NIC model. The purple arrows denote the three major data flows cross-referenced in Section 2.4.

memory-like slaves. *AXI-Lite* is designed for lightweight memory-mapped access for lower-performance situations; it does not support advanced features like bursting, narrow transfers or interleaved requests, making it a lot simpler to implement. *AXI-Stream* is designed for streaming data from master to slave without addressing semantics.

Hardware used in this thesis We use the VCU1525 Development Kit from Xilinx² for development and testing of FPsPIN. The board comes with a Xilinx UltraScale+ VU9P FPGA, 64 GB of DDR4 memory and 16 lanes of PCIe 3.0. In addition, it also has 2 QSFP+ cages that support up to 100 Gbps Ethernet; each QSFP+ port can be split into 4 25 Gbps Ethernet ports with a breakout cable³. As later to be introduced in Chapter 3 and Chapter 6, we operate the 100 Gbps Ethernet ports using a loopback cable without splitting them.

2.4 PsPIN and RISC-V

While sPIN defines the streaming in-network-computing NISA, it does not specify the exact micro-architecture of sPIN-enabled NICs such as the *instruction set architecture* (ISA) for the HPUs or the exact memory hierarchy on the NIC. PsPIN [7] is the reference ASIC implementation of sPIN, ready to be integrated into the packet data path of existing NIC designs. It specifies the interface of a NIC into which PsPIN can be integrated. PsPIN uses the

²<https://www.xilinx.com/products/boards-and-kits/vcu1525-a.html>

³<https://www.fs.com/de-en/products/70537.html>

CPU cores developed by the PULP [27] project to implement the HPUs. It groups the HPU cores into *clusters* for a hierarchical memory architecture and multi-level scheduling. An overview of the PsPIN architecture and NIC model is shown in Figure 2.4.

The control flow of PsPIN starts with new packets arriving from the NIC. The NIC inbound engine generates a *handler execution request* (HER) that contains metadata for scheduling the packet on a HPU, including the address of packet data in the NIC memory and the address of the sPIN handler functions. The *global* scheduler then resolves the scheduling dependencies according to the sPIN NISA as described in Section 2.2 and forwards individual *tasks* to the *cluster* schedulers. The cluster scheduler forwards the incoming tasks to the HPUs local to that cluster for execution, and collects the finish notifications from the HPUs called *feedbacks*. It forwards the feedback through the global scheduler back to the NIC inbound engine, such that the packet buffer can be deallocated and reused.

Three major data flows cover the full cycle of packet processing in PsPIN, all of which are driven by various DMA engines, allowing fast data movement and latency hiding. The inbound packet data from the NIC inbound engine to the L2 packet buffer is handled by the DMA engine in the NIC inbound engine; the data is further DMA'ed into the cluster-local L1 memory by the cluster DMA engine (①). Host memory access by the HPUs flow from L2 or L1 to the host memory and is handled by the *off-cluster* DMA engine (②). Outgoing packet data from L2 or L1 is handled by the DMA engine inside the NIC outbound engine (③). PsPIN exposes AXI slave ports for access to the internal interconnect by the NIC DMA engines.

RISC-V [28, 29] is an open ISA developed at UC Berkeley and now hosted by the non-profit RISC-V Foundation. It has much momentum in the community of both research groups and companies due to its free and open nature and has seen many open-source [30, 31, 27] and commercial [32, 33] implementations. The openness of RISC-V to custom extensions and the abundance of open-source implementation facilitate diverse architecture research [34, 35, 36, 37, 21] where it was previously almost impossible to build hardware prototypes due to the closed and proprietary nature of existing ISAs and expensive and restrictive licensing of processor IPs.

2.5 Corundum

Corundum [8] is an open-source, FPGA-based NIC and a platform for in-network computing. The project supports 10/25/100 Gbps Ethernet on Xilinx and Intel platforms. It offers a high-performance, custom PCIe DMA system and open-source platform-agnostic IPs including the Ethernet *media access control* (MAC) layer and AXI infrastructure. Corundum also has sup-

port for scatter/gather DMA, checksum offloading, as well as support for multi-interface multi-port operation. It offers a full software stack on Linux, exposing fine-grained scheduling and queue management tunable to the user. The comprehensive feature set and open code base make Corundum an ideal platform for high-performance network research.

The most important design of Corundum for the scope of this thesis is its support for custom hardware logic to extend the functionality of the NIC. The code base allows developers to pack custom logic into a self-contained *application block* with access to the control path, data path, and DMA subsystems. Corundum's software stack also provides kernel interfaces for custom drivers and user space utilities, as well as example designs. As a result, Corundum makes a perfect candidate platform for integrating a packet processing cluster like PsPIN to build a full SmartNIC.

Now that we have all the pieces. . .

We have now introduced the essential components towards building FPsPIN. However, even though we have the major parts of the SmartNIC ready, we still need to integrate them to create a complete SmartNIC. As we have shown in Figure 1.1, apart from the PsPIN processing cluster marked in green, we also need the data path engines and PCIe interface in hardware. Some of these components come from Corundum; others, such as the matching engine that determines which packets are going to be processed by the cluster, need to be designed and implemented from scratch. In addition, as Corundum and PsPIN are developed independently, we need various bus interconnects to bridge the control and data paths. We explain the design and implementation of these hardware components in Chapter 3. The host-side device drivers and the runtime library for interfacing with the processing cluster need to be implemented as well; we discuss this in detail in Chapter 4. We present in Chapter 5 the shortcomings of the sPIN NISA that we discovered while building FPsPIN. Finally, we explain in detail in Chapter 6 the experiments we conduct to showcase the functionality and performance of FPsPIN. Let's dive in!

Hardware

People who are more than casually interested in computers should have at least some idea of what the underlying hardware is like. Otherwise the programs they write will be pretty weird.

– Donald Knuth, *The Art of Computer Programming, Vol. I*

As we introduced in Section 2.4, PsPIN is a RISC-V-based packet processing cluster implementing the sPIN in-network-computing paradigm. However, PsPIN itself does not consist of a fully functional SmartNIC due to the lack of capability to receive and send packets; it also lacks an interface to read from and write to the system memory. The following three classes of hardware components need to be implemented to achieve full functionality of a sPIN NIC:

- the *data path*: the PsPIN cluster should be able to receive packet data from the network and send a reply back into it;
- the *control path*: the PsPIN cluster and other components should be configured from the host over various control registers and program memory (code and data); and finally,
- the *host-side DMA*: the PsPIN cluster should be able to read from and write to the main memory on the host system to establish the full sPIN programming model.

An overview of all the hardware components is shown in Figure 3.1. We now walk through the design and implementation of these modules in more detail.

Module Name	Description
pspin_host_dma	Host acdma adapter
pspin_ingress_datapath	Collective ingress data path wrapper
pspin_her_gen	HER generator
pspin_ingress_dma	Ingress DMA engine
pspin_pkt_alloc	Packet buffer allocator
pspin_pkt_match	Packet matching engine
pspin_ctrl_regs	Control registers adapter
pspin_egress_dma	Egress DMA engine

Table 3.1: Description of the modules shown in in Figure 3.1.

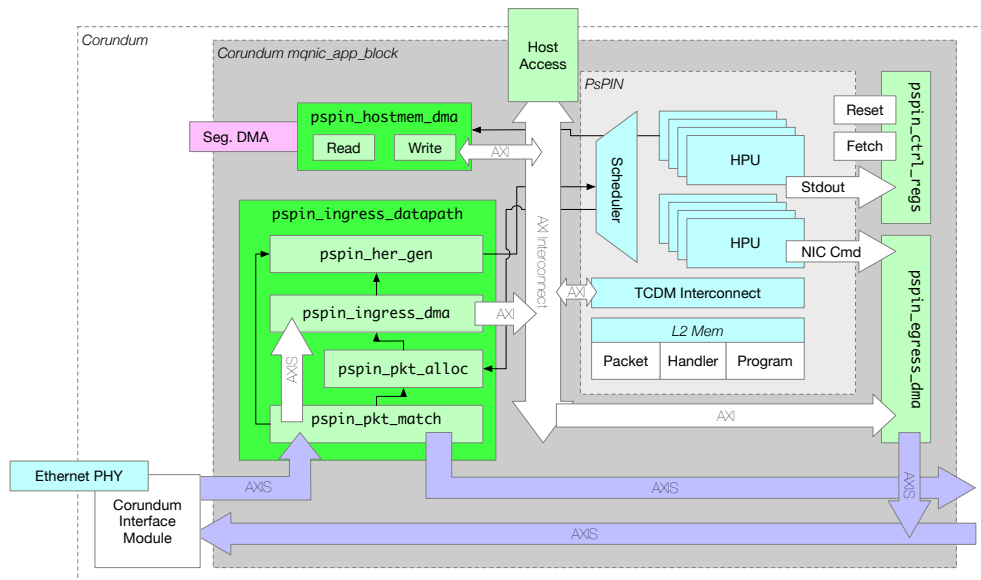


Figure 3.1: Overview of the FPsPIN hardware. A description of the hardware function blocks is shown in Table 3.1. Blocks marked in green are the modules implemented as part of this project to bridge the PsPIN cluster to Corundum.

3.1 Control Path

The control path handles configuration of the PsPIN cluster as well as the various data path components *before* the actual execution of handler code on the cluster. There are three important control-path tasks to perform from the host, all of which are implemented over Corundum’s slow-path 32-bit AXI-Lite (Section 2.3) interface with an address bus of 16 bits:

- to toggle various control registers to the PsPIN cluster and data path components;
- to read back standard output produced by PsPIN (i.e., `printf`); and

- to load program code and data onto memory in the PsPIN cluster.

Control registers The control registers are configured through the `pspin_ctrl_regs` module. The module exposes an AXI-Lite slave towards the AXI-Lite interconnect and converts this into simple `valid`-guarded interfaces (Section 2.3) for PsPIN and various data path components to consume. Some signal groups have requirements on *consistency of update*, that is, the signals in the same group should always be consistent and no partial updates should be visible to the components being controlled. Checks for this requirement happens in the kernel driver as we will introduce in Section 4.1.2. An overview of the exposed control signals from `pspin_ctrl_regs` is shown in Table 3.2.

Name	Direction	Description
<code>cl_fetch_en</code>	O	Fetch-enable control to PsPIN
<code>aux_rst</code>	O	Auxiliary reset for PsPIN and data path
<code>cl_busy</code>	I	Cluster busy status from PsPIN
<code>mpq_full</code>	I	<i>message processing queue</i> (MPQ) full status bitmap
<code>match_*</code>	O	Matching engine configuration
<code>her_gen_*</code>	O	HER generator configuration
<code>stdout_*</code>	O	Standard output readback

Table 3.2: Overview of the control wires exported by `pspin_ctrl_regs`. The meaning of these control wires will be introduced in the coming sections.

The control registers module is designed to allow reconfiguration during normal operation of the system. Therefore, components that take configuration data from the module are expected to have an explicit *valid* signal, if they expect consistency between multiple registers. The software that controls these register would then de-assert *valid*, change the registers, and then reassert *valid*, such that the downstream module can have a consistent configuration.

We group registers by the subsystem they control (e.g. the matching engine or the HER generator) and assign a block of address in the control register address space. We then refine these groups into subgroups that each of them control a specific field of configuration; some of the subgroups contain multiple identical register instances (e.g. for multiple rulesets in the matching engine). As shown in Figure 3.2, the 16-bit control register address uses the top 4 bits ("`grpid`") to address the register groups and the lower 12 bits ("`regid`") to address the subgroup and register instances. We do not explicitly define a subgroup field in the address due to different subgroup sizes across different groups.

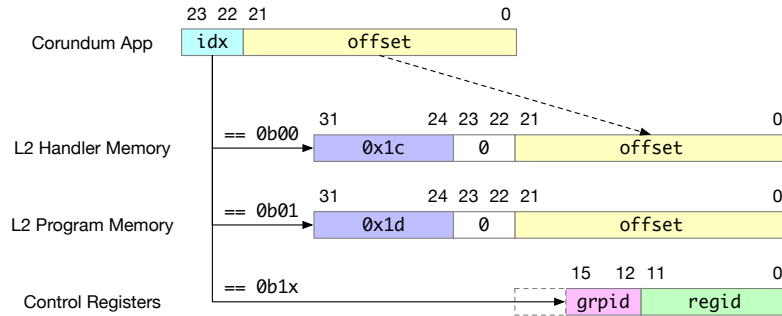


Figure 3.2: Address translation from the Corundum application control space. Access to the PsPIN host access space always have the top bit as 0; we use the second top bit to select the correct memory area in PsPIN. Access to the configuration registers have the top bit set to 1; the 22-bit offset is decoded as the 16-bit control register address and the top 6 bits ignored.

To maintain consistency about the naming, word size, and addressing about various registers, we define registers in a declarative approach with an external generator (`regs-compiler.py`). The hierarchical organisation of registers is demonstrated by the metadata describing all the registers, shown in Listing 1. The generator allocates addresses in the register address space w.r.t. the group-subgroup abstraction described in the paragraph above.

Verilog modules that interact with the control register system are written in a template language, namely Jinja [38], that abstracts the exact register definitions away. A generator written in Python processes all Verilog templates w.r.t. the register metadata and emits the final source file ready for synthesis. Such an approach eliminates the tedious and error-prone maintenance of repetitive register definitions and proved to be crucial as the number of control wires grows. The generator also derives part of the kernel driver that later exposes these control registers as described later in Section 4.1.2.

Standard output access To facilitate debugging of handler code on the PsPIN cluster, we implement a readback mechanism for the characters printed by the RISC-V cores. The core executes `putchar` to write characters into the `apb_stdout` module. Different cores write to separate addresses exported by the module, allowing the module to demultiplex the incoming characters. The module enqueues the characters together with the source core ID in a *first-in first-out* (FIFO). The FIFO is then read out from `pspin_ctrl_regs`. To avoid introducing module ports on all levels of *register transfer level* (RTL) hierarchy, we utilise the *hierarchical reference scope* [39] feature of Verilog to connect the output ports from `apb_stdout` directly. Finally, the host can read back the enqueued characters by reading out the `stdout_*` registers through the register interface, demultiplex, and store the output as logs for future inspection.

```

1 groups = [
2     RegGroup('cl', [
3         RegSubGroup('ctrl',    False, 2),
4         RegSubGroup('fifo',    True, 1),
5     ]),
6     RegGroup('me', [
7         RegSubGroup('valid',   False, 1, 1),
8         RegSubGroup('mode',    False, 4, 1),
9         RegSubGroup('idx',     False, 16, 32),
10        RegSubGroup('mask',    False, 16, 32),
11        RegSubGroup('start',   False, 16, 32),
12        RegSubGroup('end',     False, 16, 32),
13    ]),
14    RegGroup('her_meta', [
15        RegSubGroup('handler_mem_addr', False, 4),
16        RegSubGroup('handler_mem_size', False, 4),
17        RegSubGroup('host_mem_addr',   False, 4, 64),
18        RegSubGroup('host_mem_size',   False, 4),
19        RegSubGroup('hh_addr',         False, 4),
20        RegSubGroup('hh_size',         False, 4),
21        # ... more subgroups ...
22    ]),
23    # ... more groups ...
24 ]

```

Listing 1: A simplified excerpt of the metadata definition of control registers in FPsPIN. `RegGroup` defines the name and children (subgroups) of the register group; `RegSubGroup` defines the name, immutability (from the CPU), number of registers, and optionally the signal width in hardware of the subgroup. These are later consumed by Verilog and C templates to generate the register definition and use sites.

Code and data download The code and data of the packet handler program on PsPIN need to be loaded into the *program memory* in PsPIN before we can start scheduling packets to execute on the HPUs. The program memory is accessible through the *host slave* port on the PsPIN cluster. This port also allows write to the other memory area, the *handler memory*, to allow writing either static or dynamic configuration data by the host. Together, this allows loading compiled PsPIN program images onto the cluster memory.

We implement such access by connecting the upstream AXI-Lite port from Corundum, through a AXI-Lite interconnect and a AXI-Lite to AXI4 adapter, to the host slave port. Note that the PsPIN host access address space on the host slave port is 32-bits. However, we only have a 24-bit address space from the application block control port from Corundum. Therefore, we perform a *compression* in the address space by mapping the two memory areas closer together into the application control port address space; we demonstrate this in Figure 3.2. The FPsPIN kernel module (Section 4.1.2) will encode the PsPIN memory accesses according to this mapping.

3.2 Data Path

PSPIN, being a packet processor, needs to have access to the receive and transmit paths in the NIC to function properly. We introduce in this section the design and implementation of the ingress and egress data path engines that gives PSPIN access to the packet data path.

Attach points of the data path Corundum provides access to raw Ethernet frames over the AXI Stream interface. Three attachment points are available to the application block for reading ingress Ethernet frames out, as well as injecting egress frames:

- *Direct*: the AXI Stream interface directly after the Ethernet MACs and before most Corundum modules. The interfaces are synchronous to the MAC clock (322.265625 MHz for 100 Gbps Ethernet). This offers the lowest possible latency from the application block.
- *Sync*: the AXI Stream interface after the *clock domain crossing* (CDC) FIFO for each port. These interfaces are synchronous to the Corundum core clock (250 MHz). They offer comparatively low latency.
- *Interface*: the AXI Stream interface after the main packet aggregation FIFO per interface. These interfaces are per interface (instead of per port; for example a 100 Gbps interface could be split into 4 25 Gbps ports, as described in Section 2.3) and are the simplest to process. They are synchronous to the Corundum core clock (250 MHz).

The FPGA board we use, as described in Section 2.3 and later in detail in Section 6.1, has two 100 Gbps interfaces; each interface can be further split up into 4 25 Gbps ports. For simplicity of implementation, we attach the PSPIN data path at the *interface* attach point, such that we don't have to multiplex traffic from different ports by ourselves.

3.2.1 Ingress

After a packet has arrived at the *interface* attach point, multiple tasks need to be done for an ingress packet before it lands in PSPIN memory and is ready for processing. We implement four separate functional blocks as follows; together they form the ingress data path module (`pspin_ingress_datapath`):

- `pspin_pkt_match`: match if the packet is to be processed by the SmartNIC cluster or to be forwarded to the normal Corundum data path;
- `pspin_pkt_alloc`: allocate buffer for the incoming packet in the L2 packet buffer, free the buffer once it finishes processing;
- `pspin_ingress_dma`: DMA write the packet data into the L2 packet buffer

- `pspin_her_gen`: generate the HER to the PsPIN cluster

We explain in detail the design of these modules. Note that common design considerations presented in Section 3.4 apply to these modules.

Packet matching engine `pspin_pkt_match` exposes one AXI-Stream slave (`s_axis_nic_*`) towards the upstream packet data that comes from the application block interface in Corundum. It further exposes two AXI-Stream master ports towards the downstream packet processing logic. One of them (`m_axis_pspin_*`) forwards the matched packet data to the rest of the data path components for further processing. In addition, the module also exposes metadata for the matched packet over a ready-valid interface (`packet_meta_*`) providing the downstream components with the following information:

- *Message ID* from the *sPIN Lightweight Messaging Protocol (SLMP)* packet header (see Section 5.1 for details of the SLMP protocol), for the HER generator
- *end-of-message (EOM) bit* as specified by the matching ruleset, for the HER generator
- *Ruleset ID* of the matching ruleset, for the HER generator to select the correct *execution context (ECTX)*
- *Length* of the packet, for the packet buffer allocator

Since we need to count the length of the packet, the packet metadata can only be generated after that the packet has been transferred on the AXI-Stream interface. A later stage in the data path (the ingress DMA engine) will reverse this dependency by buffering the packet data.

We adopt a simple approach to define the matching rules similar to the IPTables U32 match [40]. The matching engine provides a configurable number of *rulesets*. We expose ruleset configuration to the host as control registers. Each ruleset is defined by a configurable number of *matching rules* for the *matching units*, which, each one on its own, matches against a 32-bit word of the packet and produces a boolean output. Given index I , 32-bit mask M , 32-bit start value S , and 32-bit end value E , the matching unit output is defined as:

$$\text{Output} := S \leq (\text{Packet}[4I : 4I + 3] \& M) \leq E$$

Each ruleset defines a *mode* in which the output from the matching units are combined into the match output of the ruleset. We currently implement two modes: `MATCH_AND`, which combines the match unit outputs with a logical AND; and `MATCH_OR`, for a logical OR. The module is designed such that it is

easy to add another combination mode, if such a use case rises (for example an *exactly-one* combination mode). If any of the installed rulesets matched on the packet, the module marks the packet as matched for further processing in the data path. The module then sets the *ruleset ID* metadata of the packet accordingly for ECTX selection as described later when we introduce the HER generator.

A few examples of common matching rules are as follows:

- **RULE_IP**: match EtherType at byte 12-13 = 0x0800
→ matcher #3, mask 0xffff0000, range 0x08000000-0x08000000
- **RULE_IP_PROTO**: match IP protocol at byte 23 = 17 (UDP)
→ matcher #5, mask 0xff, range 0x11-0x11
- **RULE_UDP_DPORT**: match UDP destination port at byte 36-37 = 9330 (SLMP)
→ matcher #9, mask 0xffff0000, range 0x24720000-0x24720000
- **RULE_FALSE**: match nothing ≥ 1 and < 0 (logic false)
→ matcher #0, mask 0x0, range 0x1-0x0

The other AXI-Stream master interface (`m_axis_nic_*`) performs a *pass-through* of packets that did not match with any installed rulesets back into the regular Corundum packet data path. This allows the NIC with PsPIN attached to it to still function as a normal NIC when PsPIN is not configured. It also enables host processing of traffic that is not of interest to PsPIN, for example in handling the *Address Resolution Protocol* (ARP) as described in Section 5.4, or when implementing an application-level control plane in the MPI Datatypes application as described in Section 6.3.2.

Packet buffer allocator The packet buffer allocator takes the metadata from the matching engine and allocates a buffer for the packet in the L2 packet buffer of PsPIN. It runs the allocation algorithm based on the packet length, adds the resulting address of the allocated buffer to the packet metadata, and forwards the metadata to the DMA engine to actually write the packet into the memory. It takes in the *feedback* from PsPIN, which denotes that a packet has been processed and its buffer can be freed, to free the buffer correctly. It further outputs one statistics counter of how many packets have been dropped due to the buffer being full.

The Verilator model originally developed in the PsPIN project uses a software-based ring buffer in the simulation testbench to allocate space for incoming packets in the packet buffer. The free algorithm needs to keep a queue of out-of-order frees and is thus difficult to implement in hardware. However, most packets on the Internet and in data center environments follow a *bimodal* distribution in size: 40% of packets are below 64 bytes and another

40% are 1500 bytes (the *maximum transmission unit* (MTU) of an Ethernet/IP network) [41, 42]. We thus take a simpler *fixed-size* allocation approach: we partition the packet buffer into two halves; in one half we make fixed 128-byte slots, and in the other half we make 1536-byte slots. We store these free slots in two separate FIFOs. We then handle allocation and free simply by popping from and pushing to the respective FIFOs. This way, we greatly simplify the hardware implementation of the allocator while not sacrificing too much buffer utilisation on internal fragmentation.

Ingress DMA The ingress DMA module takes the allocated address and length in the packet metadata and performs a DMA transaction to write the packet data to the PsPIN NIC inbound memory port. Upon finish of the DMA request, the module forwards the packet metadata on to the HER generator in the data path, such that the packet can get scheduled on the PsPIN cluster. We use the `axi_dma_wr` module from the Corundum AXI IP library to perform the actual DMA operation.

One complication to be handled in this module is that the matching engine could only generate the packet metadata *after* transferring the packet data on the AXI-Stream bus. This is due to a dependency introduced by needing to count the length of the message. While this is handled by introducing a shallow `axis_fifo` to reverse this dependency for the DMA module, it would introduce a per-packet latency of the number of cycles it takes to transmit the packet on the AXI-Stream bus. In addition, the module has to ensure that the DMA transfer to the PsPIN packet buffer is finished before it could issue the HER to the cluster due to the current monolithic design of the PsPIN scheduler. A possible direction of improvement is discussed in Section 7.4.

HER generator Once the packet is written to the right place in the L2 packet buffer of PsPIN, the data path can now schedule the packet for processing by issuing a HER to PsPIN. Part of the information required to generate a HER comes from the packet metadata, such as the message ID and if the packet is the last in a message (*End-Of-Message*, EOM). The rest of the HER stores the address of the handler functions that the packet should be processed with, as well as the host DMA and L2 memory regions. We expose a register control interface to the host through `pspin_ctrl_regs`.

Collective ingress data path `pspin_ingress_datapath` does not provide extra logic by itself, as it is simply an instantiation wrapper of the four data path components. It keeps the parameters in synchronisation among the data path modules and allows for one single place to pass in custom parameters. It also functions as a top module for end-to-end simulation and unit tests so that we can validate that the data path modules have consistent assumptions of how each other operates.

3.2.2 Egress

PsPIN also needs the ability to send packets into the network. This is needed to either complete a protocol by sending back acknowledgements, or transmit packets to other nodes e.g. to implement in-network AllReduce [43] with PsPIN. The transmission of the prepared egress packet is handled by `pspin_egress_datapath`; we discuss about potential problems in preparing the outgoing packet and solutions in Section 5.4.

`pspin_egress_datapath` handles egress commands from PsPIN. With the Corundum IP `axi_dma_rd`, the module performs a DMA read from the packet buffer and gets an AXI-Stream bus that contains packet data. It then injects the AXI-Stream into the outbound AXI Stream of Corundum with an AXI-Stream arbiter (`axis_arb_mux`). The arbiter is wired such that the outgoing traffic from PsPIN has priority over egress traffic from the host for maximum possible throughput from PsPIN. It can also be configured to use round-robin arbitration to ensure fairness between the host and PsPIN on outgoing packets.

3.3 Host DMA

A feature that distinguishes the sPIN programming model from other packet processing paradigms intended for intrusion detection (IDS), for example [21], is the ability of packet handlers to read from and write to host memory. Between PsPIN and Corundum, this is enabled through the `pspin_hostmem_dma` module. The module bridges the AXI master port of the PsPIN cluster to the segmented DMA interface of Corundum [44], which takes a RAM port and a separate command bus. We utilize the AXI-Stream DMA client (`dma_client_axis_source`, `dma_client_axis_sink`) from Corundum to convert the output AXI Stream bus to AXI4 channels. For write requests from PsPIN, the module first issues a DMA command to the AXI-Stream client to capture the write data in a dual-port RAM buffer (`dma_psdpram`); it then issues a command to the Corundum DMA subsystem to DMA the data from the buffer RAM to the host memory. The read process happens in the reverse order.

There are some notable limitations in this approach, namely that the adapter is not fully AXI-compliant in multiple corner cases. We do not support irregular bursts (narrow bursts or modes other than INCR), as well as interleaved read requests. Unlike AXI4, the PCIe interface also does not support arbitrary *byte enable* (BE) configurations, so we also do not support these cases. While it is theoretically possible to handle all these corner cases, it would lead to very long combinatorial paths of the resulting hardware, which would then take too much engineering effort to fix. However, these limitations are acceptable

in our use case, since the DMA bus master in PsPIN does not issue such requests.

One important corner case to implement correctly, however, is *unaligned writes*. As mandated by the sPIN specification and also as we later will see in Section 6.3.2, unaligned transfers are essential to some applications. While it is possible to implement unaligned transfers in software by reading the affected word first to compose and issue an aligned transfer, the extra memory read transactions (up to *two* extra reads for one unaligned write) would significantly hurt performance. Fortunately, the Corundum DMA subsystem fully supports unaligned transfers. As AXI4 expresses unaligned writes as aligned writes with *strobe* (byte-enable), we implement an *address recovery* procedure that calculates the original address and length from the AXI burst *strobe* (WSTRB) signal of the first and last beat in the AXI transaction. The module then issues the unaligned transfer to the client and Corundum DMA subsystem as normal.

3.4 Design Considerations

As we stated in Section 1.1, it is not a goal of this thesis project to achieve the absolute highest possible performance. The hardware implementations are thus designed with the approach of the *simplest* hardware implementation possible. This means that modules with complicated logic e.g. the host DMA engine are simple state machines without pipelining. We also do not support concurrent requests, even if the protocol supports it (in the case of AXI4 on the host DMA engine). For the purpose of a full-system demo, we argue later in Section 6.2 that these design limitations would not impact the overall system performance.

Another limitation of the hardware performance is in the PsPIN implementation. PsPIN uses the *PULP Ultra Low Power* (PULP) [27] RISC-V cores and AXI infrastructure, which are originally designed for ultra-low-power ASIC platforms. This means that they are optimised for recent ASIC process nodes and thus have long critical paths, making them not suitable for FPGA operation. While some parameter tweaking allowed us to break very long critical paths e.g. single cycle bus across the entire SoC, most components need to be redesigned to reach a higher F_{\max} on FPGAs. We discuss possible directions to a solution in Section 7.1.

The lengthy critical paths of PULP and thus PsPIN on FPGAs mean that without significant re-engineering, the packet processing cluster could only run at a lower frequency. This situation is further worsened by the area requirements of the original PsPIN design: the 4-cluster configuration that was used in the original PsPIN paper proved to be extremely difficult, if possible at all, to place and route on the FPGA device we are using. We thus

use a 2-cluster configuration with reduced memory sizes. To further resolve the routing congestion problems, we employ the incremental implementation flow provided by Xilinx as described in Section 6.1.

In contrary to PsPIN, Corundum runs at 250 MHz on the target Xilinx devices. While it is possible to retarget Corundum to run at a lower frequency, we would have to reconfigure the clock domains and validate that the resulting design still works properly; this is a non-trivial process. Instead, we opted to *only* run the PsPIN cluster and the closely coupled data path engines at a lower frequency (40 MHz for the evaluation in this thesis; more about the setup in Section 6.1). We perform CDC on the AXI-Lite and AXI-Stream interfaces with standard IP blocks from Xilinx. We isolate timing optimisation as a separate task and keep it out of the scope of this thesis due to time constraints of the project.

Software

Software is like entropy. It is difficult to grasp, weighs nothing, and obeys the Second Law of Thermodynamics; i.e., it always increases.

– Norman Ralph Augustine, *Augustine's Laws*

As described in Section 3.1, the hardware design of FPsPIN exposed all slow-path control flows to the host CPU through the `pspin_ctrl_regs`. While this simplified the hardware design by allowing us to omit a dedicated *management core* on the FPGA, the job of configuring the hardware now lands on the host CPU. In addition, we also extended the handler runtime on PsPIN to support the new hardware integration. We explain in this chapter the different software components developed for FPsPIN. Three classes of software are required for the full operation of the hardware: Linux kernel modules, user-space library and utilities, and the updated handler runtime. An overview of the software landscape of FPsPIN can be seen in Figure 4.1.

4.1 CPU Kernel Modules

Multiple approaches to access to device memory on Linux exist and most of them require some degree of kernel-level support. One approach is to expose device I/O memory access (in the case of PCIe devices, the *base address register* (BAR)) to user-space through `/dev/mem` and host memory DMA access through `udmabuf` [45]. While this approach is commonly used when developing FPGA-based accelerators in embedded environments, it introduces severe security risks due to exposing direct physical memory access to the user-space and is thus limited to embedded systems.

The other approach is to have a dedicated kernel module that interfaces with existing subsystems in Linux and does not expose unconstrained physical memory read and write (other than for diagnostics purposes). The *application*

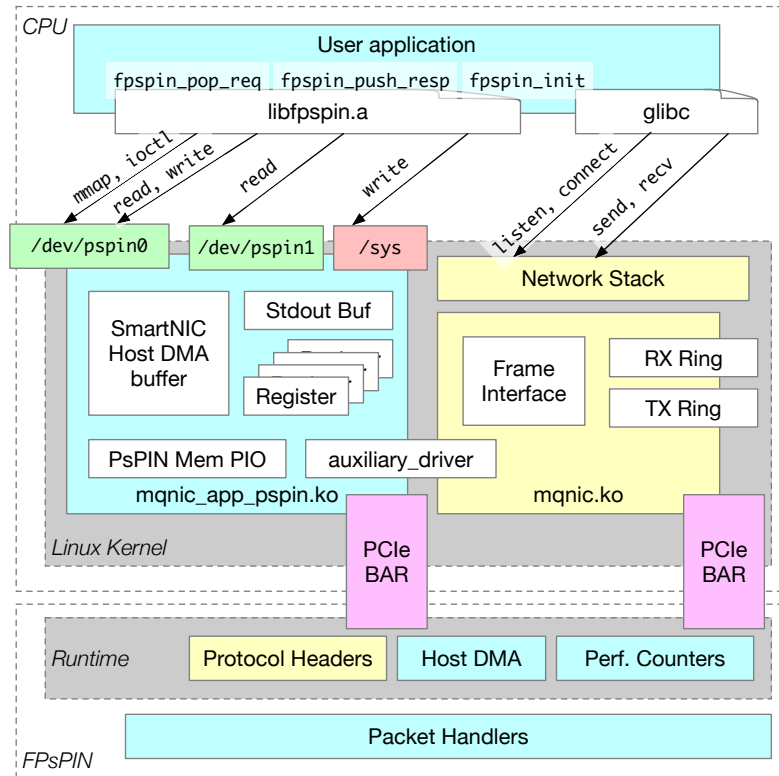


Figure 4.1: Overview of the software on the host. Yellow blocks denote existing software, while blue boxes show software developed in this project’s scope. Note that we use only standard Unix syscalls (`read`, `write`, `mmap`, `ioctl`) between the user- and kernel-space.

programming interface (API) exposed by the device driver kernel module not only greatly reduces the attack surface, but also abstracts away details of the hardware between different revisions, facilitating development of user applications and support libraries. While writing a dedicated kernel module requires experience with kernel programming, we argue that this is a necessity in hardware development for this thesis. In addition, the overhead of doing so has already been greatly reduced by Corundum from their application block driver templates. This is the approach adopted by Corundum (`mqnic.ko`) and in turn by FPsPIN (`mqnic_app_pspin.ko`) in this thesis.

4.1.1 `mqnic.ko`

Corundum ships a kernel driver for the complete NIC functionalities, including interactions with the Linux network stack to expose the device as Ethernet NICs for packet transmit and receive, as well as control interfaces with `ethtool` that reports link status. In addition, it also exposes a device file `/dev/mqnicX` for the user-space libraries and utilities to perform management

tasks, such as online firmware upgrade and device reset control.

Corundum provides driver support for the custom application block through the *auxiliary bus* framework [46] in Linux. The framework allows splitting drivers for largely independent functionalities on the same device into different device drivers and thus different modules to allow compartmentalisation and separated operation. The main device driver registers an auxiliary *device* while the sub-component driver registers an auxiliary *driver* with the framework. In Corundum, the main driver registers the application block as an auxiliary device and exposes the *application base address* (a separate PCIe BAR) to the auxiliary driver. This allows the custom driver to access the application block BAR to interact with the hardware.

4.1.2 mqnic_app_pspin.ko

The driver for FPsPIN configures the PsPIN cluster and additional datapath components after they are brought out of reset. The driver exposes two device nodes, `/dev/pspin{0,1}`, as well as a selection of device registers over `sysfs` [47]. All user-space operations during configuration and normal operation happen through access to these resources using standard *system calls* (syscalls). In addition to normal operation, the kernel module checks for additional requirements imposed by the hardware and rejects requests from the user-space that violates these requirements. We explain the main functionalities of the kernel module in this section.

Control registers The control registers from the hardware are exposed as access to the *application base address* from the Corundum auxiliary device. We use the register generator introduced in Section 3.1, `regs-compiler.sh`, to generate the respective `sysfs` node implementations; the register group and subgroup hierarchies are directly translated into *device attributes*. The generative approach keeps the driver’s view of the device registers consistent with actual hardware. We implement consistency checks of data-path engines via internal flags that are kept in sync with the respective enable registers, such that only valid and consistent configurations can be latched into hardware.

By exposing the hardware registers directly to user-space through `sysfs`, we adopt a *user-space-centric* approach to hardware configuration. This means that most configuration logic will be implemented in a user-space library (Section 4.2) instead of directly baked into the kernel module. This allows more flexibility in the implementation, since we do not need to update the kernel module as often; it acts more as a *shim* that only enforces basic safety and forwards other requests directly to the hardware. This approach also offers more protection against programming errors when implementing the configuration routines, as errors in the user-space cannot crash the kernel.

Standard output read-back Recall that as described in Section 3.1, the HPUs write their standard output into a FIFO for the host CPU to read for diagnostic purposes. The FPsPIN kernel driver exposes the `/dev/pspin1` character device to the user-space. For simplicity, the raw word sequence read from the hardware FIFO is directly exposed: each 4-byte word encodes one character as well as which HPU wrote this character. A user-space script later introduced in Section 4.2 would de-multiplex this stream and write a log file for each HPU.

In the current design and implementation, the standard output device file is *on-demand*, meaning that data will be fetched from the FIFO only when a user-space program reads from the device file. This has the potential issue of the HPUs writing too fast to overflow the FIFO, resulting in a partially lost and corrupted output buffer. An alternative design is to run a kernel *worker* (also known as a kernel thread) that continuously polls on the hardware FIFO and actively fetches the standard output data as soon as it is available. However, this would result in a constant overhead for busy polling and wouldn't be ideal if we do not care about the debug output. We thus stick to the current on-demand design.

PsPIN memory access As part of the configuration process, the host needs to download the code and runtime data for the HPUs onto NIC memory. As explained in Section 3.1, a technicality due to the small Corundum control port address space mandates a static address mapping when accessing PsPIN memory from the host. The kernel module implements this mapping and maintains the plain address view assumed by the PsPIN *software development kit* (SDK); requests that does not land in a valid memory area will be rejected with a SIGBUS (bus error signal in Linux) to avoid disrupting the hardware. We hide the translation technicality away and never expose the exact mapping details to the user-space.

The kernel module exposes two *flavours* of APIs to the user-space for accessing PsPIN memory, designed for different use cases. The first flavour conforms to the traditional non-buffered Unix file I/O: we implement the `open()`, `seek()`, `read()`, `write()`, and `close()` syscalls on the `/dev/pspin0` character device. Reads and writes to the device file are directly translated into reads and writes in the NIC memory region. This flavour is suitable for bulk read or write on the PsPIN memory area and would be used during program image load or debug memory dumping. It allows existing, unmodified Unix user-space utilities such as `dd` [48] to work as diagnosis tools and quick prototypes.

The second flavour is implemented as `ioctl()` over the `/dev/pspin0` device file. An *ioctl* (input/output control) is a syscall for device-specific I/O operations. The syscall allows the user-space application to pass a pointer

to the kernel to read or modify, along with an *ioctl number* to denote the operation desired. We implement two *ioctls*, `PSPIN_HOST_WRITE` and `PSPIN_HOST_READ`, allowing the user-space to read and write 64-bit words in one action. This simplifies the implementation of host DMA and performance counters user-space routines (Section 4.2) and reduces the syscall overhead. In comparison, the traditional Unix file I/O approach would require two separate syscalls (`seek()` and `read()` or `write()`).

Host DMA Memory pages used for DMA on Linux have to be registered with the kernel to ensure that cache coherency and alignment requirements are fulfilled. It is also important to make sure that the memory page used for DMA are not moved by the kernel through swapping or memory compaction (through `kcompactd`). The easiest way to ensure these requirements is to have the kernel module allocate the DMA buffer through the DMA API, which takes care of these requirements automatically. We implement the `mmap()` syscall for the `/dev/pspin0` device to perform a *multi-use* DMA allocation (as opposed to *single-use*; termed as *coherent* by Linux, but does not actually imply cache coherency). We then mark the area as *uncached* and map the allocated DMA memory area into the user application address space to allow user processing of host DMA traffic.

Since we adopt a user-space-centric approach regarding the configuration registers, the user-space needs access to the physical address¹ of the mapped DMA area to write to the control registers. We implement another *ioctl* on `/dev/pspin0`, `PSPIN_HOSTDMA_QUERY`, to allow the user-space to query the physical address of the DMA area, in order to program the `ECTX` to the data-path engines, specifically the HER generator.

The multi-use DMA buffer allocations we use suit the purpose of a DMA buffer shared between the CPU and device over a rather long period of time. However, in the practice of implementing NIC drivers, the *single-use* allocation scheme is more commonly used and allegedly more performant due to the possibility of taking advantage of the cache. It is possible to take advantage of this approach in FPsPIN by using a separate DMA area per *message*, as opposed to the current strategy of one area per `ECTX`. We discuss a possible implementation in Section 7.5 as future work.

4.2 CPU User-Space

The user-space software for FPsPIN caters to three distinct purposes in system operation: *configuration* of the system to bring it into operative state; *runtime* that supports the host-side application to interact with the NIC; and several

¹On a system with an *I/O memory management unit* (IOMMU) enabled, this is actually the *bus* address as seen by the DMA bus masters in the device.

utilities to aid system-wide setup as well as to perform troubleshooting. They interact with the various facilities provided by the `mqnic_app_pspin.ko` kernel module. The user-space software shipped with FPsPIN are either packaged into a static library, `libfpspin.a`, along with the header files, or as standalone programs or scripts.

Configuration The main configuration routine is packaged in `libfpspin.a` as one function: `fpspin_init`. It takes as input the device node exposed by the kernel (by default `/dev/pspin0`), the separately-built sPIN handlers image, the ID of the ECTX to use, and a number of rule sets for the matching engine. The user can either select existing rule sets that match against common protocols, e.g. TCP or UDP over IP/Ethernet, or define their own rule sets by filling in the `fpspin_ruleset_t` struct that contains configurations for each matching unit (review Section 3.2.1 for more details). `fpspin_init` configures all the device registers over `sysfs`, loads the sPIN handler image, and also allocates the host DMA area by requesting through `mmap` upon the kernel. It then fills in all necessary addresses and handles in the context variable `fpspin_ctx` and returns this to the user. All future interactions with the runtime takes the context as an argument.

After a successful return of `fpspin_init`, FPsPIN is ready for packet processing. However, in complicated applications e.g. the datatypes demo shown in Section 6.3.2, the user may wish to perform additional initialisation, e.g., loading dynamically generated data into the NIC memory. This is accomplished via the host access to NIC memory interfaces provided in `libfpspin.a`, namely `fpspin_write_memory`, allowing the host to generate the NIC memory content *in the host* application at runtime. The host application needs to take care of *relocation* so that data structures contain valid NIC pointers when they are accessed by the HPU in operation.

While the basic initialisation via `fpspin_init` programs the matching engine as the last step such that no packets can arrive at the cluster until it is fully configured, host-side user initialisation happens after the HPUs have started execution. As a result, the user needs to ensure that the sPIN handlers do not start processing packets until the host initialisation process is finished, e.g. through a flag that gates all HPUs from running. The exact mechanism and interface requirements are further discussed in Section 5.5 as a possible extension to the sPIN specification.

Runtime If the sPIN packet handlers programmed to the cluster requires interaction with the host, e.g. to forward a processed incoming message to the host for further processing, the host application should then *poll* the notification interface from time to time. We currently implement a simple flag-based notification method as shown in Figure 4.2: both PsPIN and the CPU writes *remotely* and polls *locally*. The host application tries to pop a

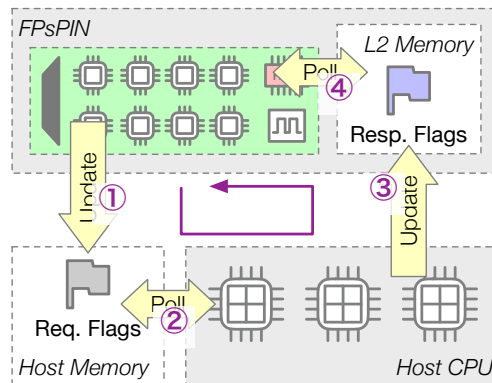


Figure 4.2: Simplified view of the host DMA loop, in chronological order. ①) The HPU sends a request to the host for processing, by writing to the flag in host memory; ②) the host polls and pops the request from local memory; ③) the host pushes the response by writing to the flag in NIC memory; ④) the HPU polls and pops the response from local memory.

notification from PsPIN using `fpspin_pop_req` (②). If a message is present, the host application processes the message and sends back a response via `fpspin_push_resp` (③). The rest of the process (①, ④) happens in the handler runtime to be introduced in Section 4.3.

It is important that the host CPU should be able to perform other workloads, such as computational tasks, during packet processing in a truly *offloading* manner. The host application can overlap other workloads via multi-threading or by anticipating the *inter-message gap* (IMG) and polling only when there could be a message arriving. For simplicity, the current flags-based host DMA notification facility can only hold one in-flight message between the host and each HPU; this limits the duration of overlapped workloads between polling to be one IMG. The overlap can be increased by implementing a proper ring buffer for the notification, which we leave as a possible future improvement.

The host application may still need to receive and send network packets on the same interface, for example to implement the slow, non-performance-critical paths of a network protocol, like connection setup and tear-down in TCP. The intended operation for this purpose is via the host network stack, either normally or through the *raw* sockets (in case of state confusion due to partially offloaded messages). The user needs to correctly configure the matching engine, so that these packets are actually delivered to the host CPU and not to PsPIN. Alternatively, if it is difficult to express the criteria in the matching rules, the user can make the handlers perform a *secondary match* and deliver such packets to the host over host DMA.

Performance measurements are important to estimate bottlenecks of packet processing. The runtime provides facility to read and clear *performance counters* exposed by the handlers. Up to 16 32-bit counters are accessible from

the host application via `fpspin_get_counter` and `fpspin_clear_counter`. Each counter keep track of a total sum and iteration count of updates, enabling the calculation of an average value. The counters are updated in the packet handlers using a facility in the handler runtime.

Utilities In addition to the `libfpspin.a` library to be statically linked into the user application, we also provide several standalone utilities that are important to the normal operation of FPsPIN. One of these is `cat_stdout.py` that reads from the log facility, `/dev/pspin1`, provided by the application kernel module. The script performs *blocking read* on the log device and demultiplexes the stream of printed characters according to the core ID. The user can specify whether to dump the log to files and if the script should remove stale logs. The script is provided separately instead of integrated into the runtime, in case of an application that does not care about the debug output from the HPUs and thus does not want to waste CPU cycles to read them.

During the development and testing of handlers, it may be necessary to read or write specific memory locations in the NIC L2 memory. The `mem` utility takes a NIC address and performs a 64-bit read or write command over the `ioctl` interface provided by the kernel module. It is possible to implement a more complicated debugger protocol with memory access in this fashion; we leave this as future work.

4.3 Handler Runtime

The PsPIN project provided a rather comprehensive implementation of the handler-side sPIN API through the PsPIN/PULP runtime. This includes the HER and task data structures, as well as host DMA commands for the handler code to invoke. A few additions are made to accomodate new abstractions introduced by FPsPIN. One of such additions concerns packet header processing. The existing PsPIN runtime already provides C structs for interpreting headers for IP and UDP, but since FPsPIN directly receives Ethernet frames instead of the IP payloads of a lower-level messaging network layer e.g. *IP over InfiniBand (IPoIB)*, we added the Ethernet header and MAC address structs for this situation. We also implement support for the SLMP, introduced later in Section 5.1, in the same manner in the FPsPIN runtime.

As we have seen in Figure 4.2, the handler issues notifications to the host for DMA events and pops the response of the host. The handler, through the runtime function `fpspin_host_req`, issues the host notification via a host DMA write (①) and polls for the host response in local memory (④). The location of the host flag sits at a pre-determined offset in the host DMA area, while the NIC flag is exposed via a global symbol in the handler image,

retrieved during loading. Note that we adopt a *synchronous* design here in contrast to the *asynchronous* design for the host-side counterpart of the host DMA process. This is currently justified since most of the host notifications happen at the end of messages in the *tail* handler and the handler would not have meaningful workload to overlap with.

Performance measurements The host-side runtime has support for reading back performance counters generated by the handler routines; these counters are updated through the handler runtime on the HPU. Each 64-bit counter consists of two 32-bit fields, the *sum* and *count*, allowing the handler logic to push a specific performance value into the counter with the `push_counter` routine. Every time the counter is incremented, the count field is incremented by one. The counters sit in L2 memory accessible to the host and are initialised to zero on cluster setup. These counters can be used to collect various statistics such as handler execution time at handler or message granularity, as well as to profile specific code areas in the handler routines.

The counter values for performance measurements are derived from the *cycles* register in PULP that is only accessible to *machine-mode*, while the handler code executes in user-mode. We extend the existing fault handler in the PsPIN runtime to handle syscalls and implement a syscall to read the cycles register. With the current naïve implementation, this syscall path takes around 100 cycles on the PULP cores, in which most of the time consumed comes from the need to save and restore all general purpose registers. While it is possible to optimise this path for a lower latency, a proper solution is to address the lack of a user-accessible time register for precise performance measurements as we will argue in Section 5.2. We leave the related changes to hardware as future work.

sPIN Revisited

Their guess turned out to be right, but one is reminded of E. T. Bell's remark that the great vice of the Greeks was not sodomy but extrapolation.

– John Drury Clark, *Ignition!: An Informal History of Liquid Rocket Propellants*

While we started this thesis to build an FPGA-based sPIN-compliant NIC, the process of building FPsPIN has uncovered various aspects in the sPIN specification that are important in a real-world system but left unspecified. We have reported the findings in this chapter to the sPIN team and some of them have already been incorporated back into the specification.

5.1 Messaging and Reliability Layer: SLMP

The sPIN specification did not impose a fixed list of underlying network protocols; instead, it specifies two *matching modes* of the underlying network. In *packet matching*, single packets are matched for processing on the packet handler in the same flow; Ethernet would be an example of a network operating in this mode. *Message matching*, on the other hand, requires the network to provide an abstraction of messages as a stream of multiple packets; they are in turn mapped onto the *head*, *packet*, and *tail* handlers for processing. Examples of a network operating in message matching mode are RDMA-style networks such as InfiniBand or the Intel *Omni-Path Architecture* (OPA).

5.1.1 Motivation

Although FPsPIN is built on Ethernet, which would seemingly force a *packet matching* implementation, many applications still benefit from the message-oriented abstraction sPIN offers; there would be significant HPU and memory overhead to perform flow matching and differentiate the handler code paths

in software, as opposed to the *message processing queue* (MPQ)-based hardware flow matching mechanism introduced in PsPIN. As a result, it is desirable to *emulate* the message abstraction on top of Ethernet. In addition, since Ethernet is a *lossy network*¹ but traditional high-performance network applications such as MPI expect that messages do not get lost in the network, we also need guarantee on reliable delivery of messages on top. We formalise the requirements of a suitable protocol in the situation for sPIN:

- ① **Arbitrary length messages:** the message matching mode in sPIN requires messages longer than one MTU.
- ② **EOM feature in packet header:** sPIN defines a *tail* handler for messages; this is required for a simple matching engine implementation.
- ③ **Out-of-order segment delivery:** sPIN does not specify a segment delivery order requirement; in fact, in-order delivery will create *Head-of-Line* (HoL) blocking issues that would hurt performance.
- ④ **Pluggable reliability:** high-performance networking applications require reliable packet delivery; however, since sPIN does not enforce lossless packet delivery, the reliability module should be pluggable and not forced on all applications.
- ⑤ **Simple implementation:** the on-path nature of sPIN mandates a simple design that is easy to implement and does not add too much overhead, especially on the receiver side that is offloaded.

A straightforward approach would be to adopt an existing protocol and implement it in sPIN handlers; candidates for this purpose include TCP [49], UDP [50], *Stream Control Transmission Protocol* (SCTP) [51], *Datagram Congestion Control Protocol* (DCCP) [52], and the more recent QUIC [53]. We show in Table 5.1 that these protocols does not satisfy the aforementioned requirements of a messaging and reliability layer for an Ethernet-based sPIN NIC such as FPsPIN, as well as the proposed solution in this thesis for comparison.

5.1.2 The solution

We developed a thin messaging and reliability layer built on top of UDP/IP for FPsPIN, which we named *sPIN Lightweight Messaging Protocol* (SLMP). The protocol features a 10-byte header inside the UDP payload with the following field definition:

¹While *RDMA over Converged Ethernet* (RoCE) does provide a lossless guarantee on top of Ethernet, it is not supported by Corundum at the time of this thesis and is not trivial to implement in hardware. We thus consider RoCE irrelevant for this discussion.

Protocol	①	②	③	④	⑤
UDP	X	-	-	X	✓
DCCP	X	-	-	● ¹	✓
TCP	✓	✓	X	● ²	X
SCTP	✓	✓	✓	✓	X
QUIC	✓	● ³	✓	● ²	X
SLMP	✓	✓	✓	✓	✓

Table 5.1: Comparison of different protocols for reliability and messaging on FPsPIN. ✓: fully supported; ●: partially supported; X: not supported. Notes: (1) Only for *acknowledgement* (ACK) segments and not data segments. (2) Reliability layer cannot be disabled. (3) QUIC requires all packets to be encrypted, meaning that a simple matching engine could not recognise EOM without decrypting.

- **Flags** (2 bytes): hosts three packet-level status bits, *synchronisation* (SYN), *acknowledgement* (ACK), and *EOM* (②). The remaining bits are reserved for future versions of the protocol.
- **Message ID** (4 bytes): unique ID of the message.
- **Offset** (4 bytes): byte offset of the first payload byte in the message.

The message ID and offset fields together implement ① up to a message size of 4 GB, limited by the 4-byte offset field in the SLMP packet header. We believe that this is a sensible limitation and most applications would not generate messages larger; for applications that require larger messages, we could adjust the size of the offset field in a case-by-case fashion. Although the offset field maintains a order among all the segments of a message, we do not implement any in-order delivery guarantees (③).

A possible alternative for the offset field is to use a packet sequence number instead of a byte offset. This is a design choice made to accomodate the SLMP file transfer and MPI datatypes applications we will introduce in Section 6.3; these applications process incoming segments according to their byte offset in the whole stream. Therefore, by storing the offset number directly in the SLMP header, we eliminate the need to keep protocol states on the receiver, allowing a fully stateless receiver for handling the protocol (⑤).

We handle the pluggable reliability requirement (④) through the SYN and ACK bits in the flags field in the SLMP header. The action rule for the receiver is simple: each packet that has a SYN bit set in the header needs to be ACK'ed by sending back the same header with no payload. The sender decides on what reliability mode the protocol operates in. For no guarantee at all, the sender omits the SYN bit for all packets. For a guarantee of message delivery but not individual segments, the sender sets SYN on the first and last packets of the message. For a guarantee of every single segment, the sender

sets SYN on all packets it transmits. We do not implement retransmission for SLMP at the moment for simplicity, but it should be easy to add since our ACKs carry the message ID and segment offset and thus would allow the sender to identify a lost segment.

5.1.3 SLMP flow control

If the sender would transmit packets too fast to the receiver, the receiver would be overwhelmed by incoming packets before it had time to process them; packets would be dropped once the receive buffer is completely filled. *Flow control* throttles the sender to make sure that the receiver is not overwhelmed. Generally speaking for maximum throughput, the sender needs to first fill up the receiver's buffer at a higher send rate, then lower the rate to the processing speed of the receiver to maintain the occupation rate of the receiver buffer in order to saturate the receiver's processing power.

There are two modes of flow control in SLMP, depending on the reliability configuration selected. When the sender operates without reliable packet delivery in the form of ACKs from the receiver, a heuristic *inter-packet gap* (IPG) is chosen based on the workload, receiver's capability, as well as possible buffer state reports from the receiver (see Section 5.2 for more details). This form of flow control requires almost no collaboration from the receiver and can be implemented fully on the sender side. However, a practical configuration would fix the IPG and thus the sending rate, which must be equal to or lower than the receiver processing speed for long term stable operation; this would under-utilises the receive buffer and result in the receiver waiting for data, hurting throughput.

A well-known mechanism that originated from TCP but found its way into most flow control schemes is a *flow control window*². The sender maintains a fixed window of packets that has not yet been ACK'ed by the receiver and would only send when the window has free space to fit a new packet, allowing the sender to automatically throttle down to the speed of the receiver after the window is filled. Assuming constant sender and receiver speed v_{Send} and v_{Recv} , we can calculate the ideal window size S_{Wnd} for a receiver buffer size S_{Recv} :

$$t_{\text{Fill}} = \frac{S_{\text{Recv}}}{v_{\text{Send}} - v_{\text{Recv}}} \quad (5.1)$$

$$S_{\text{Wnd}} = t_{\text{Fill}} \cdot v_{\text{Send}} \quad (5.2)$$

$$= S_{\text{Recv}} \cdot \frac{v_{\text{Send}}}{v_{\text{Send}} - v_{\text{Recv}}} \quad (5.3)$$

²This is called the *sliding window* in TCP due to the additional in-order delivery requirement.

The sender window approach to flow control still requires the window size to be determined in some manner. For simplicity in implementation, we assume the suitable window size is relatively fixed for each SLMP conversation and let the user specify the window size manually. We discuss a possible future direction of *adaptive* window selection in Section 7.3.

An interesting side-effect of allowing explicit control of the flow control window size is that, by setting the window size to 1 packet, the sender can serialise packet processing on the receiver side, only sending the next packet of the message after receiving ACK for the previous one. While doing this severely limits the top throughput available, the serialisation guarantee is important since sPIN did not specify any concurrency control mechanism on the scheduler level (discussed later in Section 5.3). We use this method to avoid *packet-level parallelism* for the MPI Datatypes demo application in Section 6.3.2.

5.2 Telemetry

The ability to measure the performance of a system is crucial to further improve it. While the current sPIN specification formalised performance-related events to be delivered to the host for further analysis, this is vastly different from the common practice of implementing *performance counters* for detailed system-level inspection. Therefore, we call for a general interface of host access to various implementation-defined performance counters in hardware, as well as user-controllable counters updated from the handler software.

We integrated a prototype of memory-backed general purpose counters for packet handlers in Section 4.2 to allow intrusive inspection of handler performance in the ping-pong demo in Section 6.3.1. For low-overhead updates to these user-controllable counters, an implementation should make accurate cycle counters available to user-level handler code. While we recognise the exposure of accurate timing information is a break of isolation, the current sPIN specification does not handle multi-tenancy yet; we leave this discussion for a possible future work that would explore operating system paradigms on sPIN.

Another compelling use case for telemetry data apart from off-line host performance analysis is for consuming in the sPIN NIC itself, better known as *introspection*. One use case would be for the scheduler to have access to the handler execution time counter for fair scheduling between multiple ECTXs. Another possible use case is for a dedicated management core for handling L3 protocols (Section 5.4) to implement explicit buffer state notification as described in Section 7.3.

5.3 Scheduler Concurrency Control

The current sPIN scheduler enforces the dependency between packets in a message w.r.t. the three categories of packet handlers: the *head* handler is guaranteed to be scheduled before all *packet* handlers, and the *tail* handler is guaranteed to be scheduled after them; the packet handlers will be scheduled in parallel if possible. In some use cases however, the packet processing routine may require serial execution; the MPI Datatypes demo application we will introduce in Section 6.3.2 is an example.

With the current sPIN specification, the only viable synchronisation primitive is *spinlocks* that would allow one HPU into the critical section for serial processing. This is far from ideal, since without an effective task switching method, all other HPUs will busy-loop at the spinlock, effectively reducing the number of HPUs down to one. A workaround currently used by the MPI Datatypes demo in Section 6.3.2 is to force the SLMP flow control window to 1 packet, effectively requiring ACK on every packet and thus serialising packet processing. Such a workaround is still not ideal, since such a small window size would mean that the HPU would always have to idle for one *round-trip time* (RTT) plus the sender latency for one packet before it can start processing the next packet. It also forces the developer to put the per-message state in the *globally-shared* L2 memory, since there is no guarantee which HPU the next packet will be scheduled to, making it impossible to use the *cluster-local* L1 memory.

We propose the addition of *core masks* to all sPIN ECTXs for fine-grained control on the locality and parallelism of sPIN handlers. With the current FPsPIN architecture, the core masks are installed into the HER generator and copied into the HER to the scheduler. The scheduler can then schedule the packet on the subset of cores specified by the core mask in the HER. This design can support the use case of serially scheduled handlers by programming a mask that contains one single core; *message-level parallelism* can be achieved by installing multiple otherwise identical ECTXs that have different core masks. Another possible use case is to specify a core mask of all cores in one cluster to allow the shared states to be stored completely in the cluster-local L1 memory.

5.4 Network-layer Protocol Handling

So far, sPIN assumes that the packet handlers only process the application payloads carried by the underlying network protocol. However, as we have shown in Section 5.1, extra protocol-layer processing is required for a lossy underlying network like Ethernet. While in some situations protocol handling itself is the main offloaded workload (e.g. accelerating QUIC), we recognise that most of the sPIN applications care more about the actual payload instead

of details in the protocol itself. Examples of such protocol-level handling include running ARP for outgoing packets, handling connection setup and teardown in TCP, and sending the explicit buffer state messages in SLMP.

One approach to this issue makes use of the *bypass* feature of the matching engine. The user can configure the matching engine to pass incoming protocol control packets to the host for handling, possibly updating relevant states in the meantime. Examples where this approach would work include ARP responses, in which the host updates the IP neighbour table and sends back the ARP response, and TCP connection setups and teardowns, where the host handles packets that have the SYN or FIN bit set. This approach would not work very well for NIC-initiated actions (e.g. an active ARP *query* for an outgoing packet from the sPIN NIC) as well as protocol messages on the hot path (e.g. the buffer state notification of SLMP).

An alternative approach is to introduce a dedicated system-level coprocessor for handling such protocol requests. This coprocessor would handle network-layer control-path tasks, freeing the HPUs from these. It would take requests from the HPUs through a mailbox-like interface, e.g. to run an ARP query or setup a SLMP or TCP connection with a remote endpoint. It could also run local periodic tasks such as sending back telemetry data to its link partner, or take action on specific protocol control messages forwarded from the scheduler. The coprocessor would also be crucial for potential *flow spilling* to the host when the sPIN NIC is overloaded.

5.5 Handler Initialisation

Complicated applications may require dynamic initialisation of application-level states on the NIC memory. However, the current sPIN specification would start scheduling packets to ECTXs as soon as they are installed, resulting in a race condition. However, since the installation of an ECTX on the host would also allocate NIC memory windows and set up memory protection, it is required before the host can actually perform initialisation.

A potential solution is to separate the *installation* and *activation* of ECTXs. The installation of an ECTX would arm all relevant hardware modules, allocate memory windows, and set up memory protection; the activation actually enables the respective matching rule on the matching engine for packets to arrive and get scheduled. The actual initialisation can happen on either the host (through NIC memory access) or as a special function in the handler image to run on a HPU.

5.6 Host-side Activation

So far sPIN has only been formalised for offloading packet processing tasks on a receiver; although [4] contemplated a possible sPIN-Out implementation that would allow the host to offload sending tasks e.g. in *AllReduce* to a sPIN NIC, the specification did not formalise on how such an ECTX would be defined. We propose *host-activated* ECTXs that are triggered via a special host-initiated event. With the FPsPIN architecture, this can be implemented via adding extra configuration registers to the HER generator to inject a HER whenever the host wants to invoke the ECTX. To avoid confusion, such an ECTX should always have the corresponding matching rule set to *never* such that it never gets invoked from incoming packets.

5.7 Alternative Host DMA Interface

sPIN specifies the host DMA facility in a very simple manner: the HPUs can read from or write to the exposed flat memory window. While this abstraction is concise to implement and allows , it is not very helpful for end users of a sPIN NIC; they would have to implement their own interface on top of this facility. The fact that sPIN does not specify any memory consistency semantics on the host memory window makes any user implementations non-portable and thus impossible to work with other sPIN implementations.

We developed a simple request/response interface on top of the host DMA window in FPsPIN as we described in Section 4.2; the current design largely resembles a traditional *queue pair* design, allowing the HPUs to post requests in the *request queue* (RQ) to the host and the CPU to post responses back to the HPUs in the *completion queue* (CQ). We propose the addition of higher-level APIs for communication between the host and sPIN NIC, which would simplify user programs that make use of host DMA and improve portability.

Evaluation

Despite the limited scope of that experiment, giving parents greater choice had a major effect on education quality.

– Milton & Rose D. Friedman, *Free to Choose: A Personal Statement*

In this chapter, we evaluate the software and hardware implementation of the proposed FPsPIN platform. We first describe the platform we implemented FPsPIN on. We then present an analysis of the hardware components proposed in Chapter 3 to identify potential bottlenecks in the hardware design and implementation. Finally, we demonstrate the overall functionality and performance of the system through several demo applications.

6.1 Experiment Setup

The experiments are done on the AMD server with the Ryzen 7 2700 CPU and the PCIe-attached Xilinx VCU1525¹ Development Kit; more details can be found in Section 2.3. We run the FPGA board at 16 lanes of PCIe 3.0 clocked at 8 GT/s. A diagram of the experiment platform is shown in Figure 6.1. Corundum runs at its native frequency of 250 MHz on the Virtex UltraScale+ FPGA. However, we could only run the application block (FPsPIN) at 40 MHz due to the PsPIN IP not being designed for FPGAs: the PULP RISC-V cores are designed for an advanced ASIC process node and have long critical paths on FPGAs. As a result, we clock the processing cluster at 40 MHz.

Networking The two 100 Gbps QSFP Ethernet ports on the FPGA board are attached via one *direct-attached copper* (DAC) cable, forming a loop-back between the two interfaces of Corundum. Since the two interfaces are present on the same Linux host, we have to isolate the two network interfaces

¹<https://www.xilinx.com/products/boards-and-kits/vcu1525-a.html>

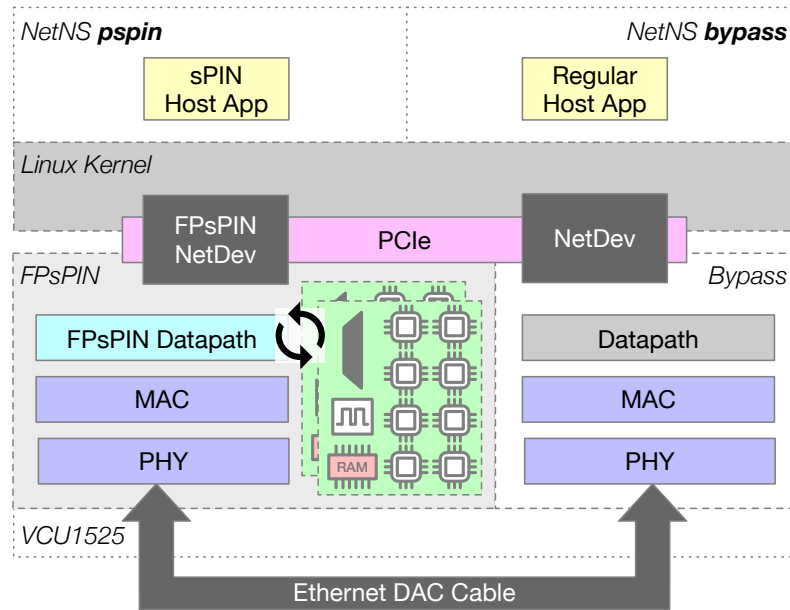


Figure 6.1: The experiment setup. The sPIN and non-sPIN host applications are in two separate network namespaces to prevent the direct loop-back mechanism in Linux that prevents packets from actually going through FPsPIN.

into separate *network namespaces* to avoid a direct loop-back in software. The exact network topology of the system can be seen in Figure 6.1. A script, `setup-netns.sh`, automates the creation and tear-down of network namespaces and assignment of the interfaces to them.

Toolchain We use Ubuntu 20.04.4 LTS on the host with a slightly modified Linux 5.15.0-76-generic kernel with the *Contiguous Memory Allocator* (CMA) enabled; this allows the FPsPIN kernel driver to allocate arbitrarily large contiguous DMA areas, as is required by demo applications shown later in Section 6.3. We use Xilinx Vivado 2020.2 to produce the FPGA bitstream, the PULP RISC-V toolchain² to compile the sPIN handlers, and the Ubuntu system GCC for the host-side applications.

6.2 Design Analysis

To evaluate the implementation quality of the newly introduced data-path components, we estimate the theoretical latency of these components as described in Chapter 3 based on the RTL source. Table 6.1 shows the latency in cycles based on the state machine construction in the Verilog RTL code, the frequency, and the latency time in nanoseconds. The `pspin_ingress_dma`

²<https://github.com/pulp-platform/pulp-riscv-gnu-toolchain>

Module	Cycles	Frequency (MHz)	Latency (ns)
Matching engine	4	40	100
Allocator	0	40	0
Ingress DMA	8~70	40	200~1750
HER generator	0	40	0
Host DMA	<i>n/a</i>	250	~450

Table 6.1: Latency estimation for various data path modules in cycles and nanoseconds. Note that as the host DMA goes over PCIe to the host DRAM, the exact latency in cycles is difficult to estimate; the latency in nanoseconds is measured on real hardware via the *Integrated Logic Analyzer* (ILA) on Xilinx platforms.

Resource Category	[7]	FPsPIN
Clusters	4	2
#MPQ	256	16
L1 Cluster Memory	1 MiB	256 KiB
L2 Program Memory	32 KiB	32 KiB
L2 Packet Memory	4 MiB	512 KiB
L2 Handler Memory	4 MiB	1 MiB
L2 SRAM Latency (cycles)	1	1

Table 6.2: Comparison between the stock PsPIN configuration and that used in FPsPIN. The MPQ enables parallel in-flight messages; by reducing the number of queues available, we limit the number of concurrent in-flight messages to 16.

module has a latency linearly related to the packet size due to dependency requirements as introduced in Section 3.2.1. We show in the later sections that these latency numbers are negligible compared to other parts of the system and thus would not have a big impact on overall system performance.

Resource utilisation and timing are very important static insights into FPGA designs. While we have trimmed the original PsPIN design significantly compared to the standard configuration [7] as shown in Table 6.2, the design is still very hard to close timing due to congestion issues. We present in Table 6.3 data in resource utilisation, timing, and time taken to implement the design. To ensure that we get acceptable implementation results for each run, we employ the *incremental implementation flow* [54] from Xilinx to have the EDA tool try to reuse routed nets from previous valid implementation runs. This shortens implementation time and improves the general *quality of results* (QoR) of the resulting design.

QoR Metric	Value	
LUT	645k	54.5%
FF	490k	20.7%
BRAM	1141	52.8%
URAM	206	21.5%
WNS (ns)	-0.057	
TNS (ns)	-9.945	
Impl. Time	6:11:15	

Table 6.3: QoR metrics of the hardware implementation of FPsPIN. The first four entries (LUT, FF, BRAM, URAM; refer to Section 2.3) denote key resource consumption and the percentage utilisation value on the VU9P device; WNS and TNS measure how much the design has failed timing.

6.3 Demo Applications on Real Hardware

We present three E2E demo applications to showcase the real-world programmability and performance of FPsPIN. We show that it is possible to write arbitrary packet-processing applications for the platform with the first two applications, *ping-pong* and the SLMP file transfer. We further characterise the performance of the platform in detail with the MPI Datatypes demo.

6.3.1 Ping-pong

Motivation We demonstrate the overall system functionality with two classic types of *ping-pong* protocols: *Internet Control Message Protocol (ICMP)* and *UDP*. With this demo, we exercise the various data-path and control-path components newly introduced in FPsPIN to show their basic functionality. In addition, we evaluate the system E2E RTT under simple packet processing workloads to compare with pure-CPU processing. We further identify RTT contributions from different actors in order to evaluate bottlenecks in the system.

Experiment We implement on FPsPIN the server to respond to client requests; the operation flow of the server is shown in Figure 6.2. Both protocols operate in the same way that the client sends a request packet and the server sends back a response. The server needs to swap the source and destination addresses in the Ethernet, IP, and UDP headers, and recalculate relevant checksums. For each protocol, we implement three different modes of operation:

- the *baseline* a.k.a. host-only case (**Host**): all processing on the CPU by setting the FPsPIN matching engine to bypass mode;

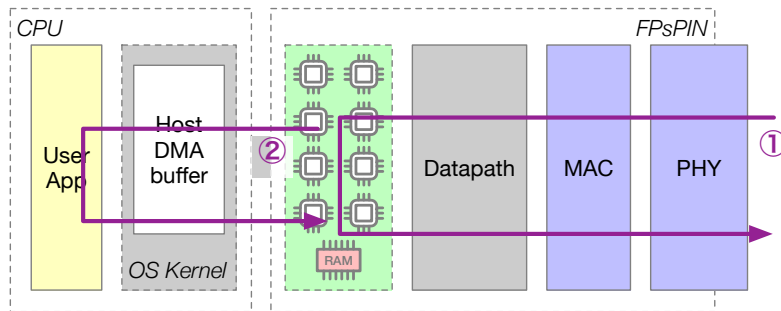


Figure 6.2: Workflow of the two ping-pong applications. ① Normal operation; the cluster processes incoming ping requests and sends back responses (**FPsPIN**). ② The cluster can optionally choose to forward data to the host application for further processing (**Host+FPsPIN**).

- the *FPsPIN-only* case (**FPsPIN**): FPsPIN does all the packet processing (header processing and checksum calculation);
- the *combined* case (**Host+FPsPIN**): FPsPIN swaps the addresses in the headers and the host CPU calculates checksums.

We use the ping utility from *iputils* [55] for ICMP and *dgping* from the *stping* suite [56] for UDP. For **Host** mode, we use the responder in the Linux kernel for ICMP and the user-space *dgpingd* for UDP. For both the **FPsPIN** and **Host+FPsPIN** modes, we use the same naive IP checksum algorithm implementation.

An important difference between the UDP and ICMP ping protocols is that ICMP requires the *entire payload* to be included in the calculation of the checksum field, while UDP only specifies an *optional* checksum of the *UDP header*; we omit this header checksum in our UDP ping server implementation on FPsPIN. This difference between the two protocols impacts both the server and client implementation, but is especially significant for the server since the RTT measurements taken at the client do not include packet preparation and checksum validation time on the client side.

For both ICMP and UDP and the three modes of operation, we measure the E2E RTT of the ping-pong process from the client by running the ping program 20 times, taking 100 measurements in each iteration. This would take into consideration any possible interference between the ping client and server that would result in variance in the measurement, as well as to allow caches to warm up. We plot this E2E RTT with their medians and 95% confidence intervals calculated with the bootstrap method [57] in Figure 6.3. In addition in cases that involve FPsPIN, we measure cycle counts in the handler code to time the mean handler execution time and latency of host processing. We plot a breakdown of the E2E RTT in these cases in Figure 6.4. Please note that the high overhead designated as *Syscall* is due to the lack of a cycle-count register accessible to user-space handler code; we discussed

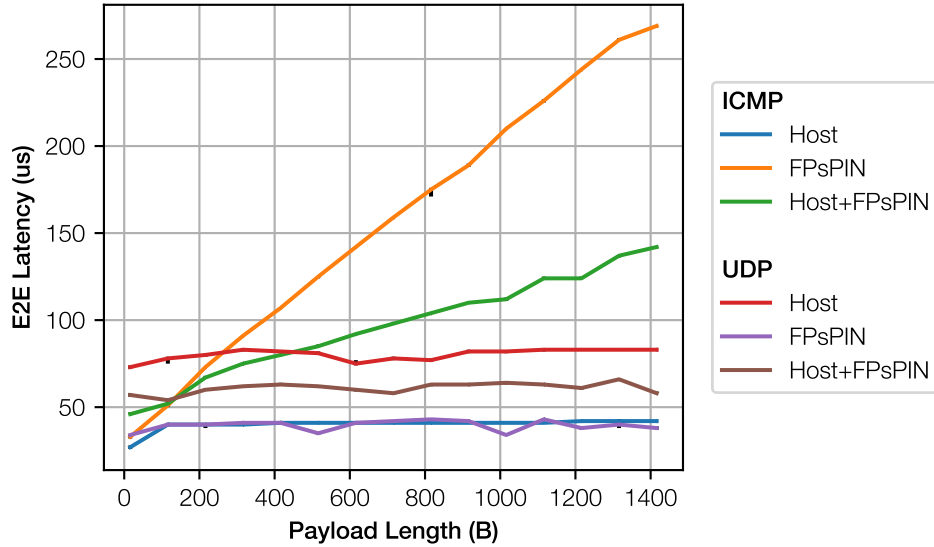


Figure 6.3: E2E RTT of both protocols across the three setups. We plot the median and 95% confidence interval of the RTT of 20 measurements for each configuration.

this situation and possible solutions in Section 4.2 and Section 5.2.

Data interpretation A key observation we make is that both **FPsPIN** and **Host+FPsPIN** performed significantly better than **Host** for UDP, with a largest latency advantage of **50 us** in **FPsPIN** mode. This is mainly due to the packet data in FPsPIN modes not having to go through PCIe to get DMA’ed to host memory, go through the Linux UDP network stack, and context switch to user-mode to reach the UDP responder. The ICMP responder in **Host**, in comparison, runs in the Linux kernel and thus does not have the overhead from the full UDP stack and context-switch to user-mode. This overhead can be confirmed with a comparison between UDP and ICMP in **Host** mode, showing a difference of ~ 41 us. As we see in Figure 6.4 in **Host+FPsPIN** for UDP, our system reliably achieves a ~ 20 us RTT advantage over the baseline case even with the added latency from host processing. In addition, we notice that in all four cases with FPsPIN the *syscall* category occupies 10~20 us in the RTT. As we have previously explained in Section 4.3, this added latency is due to a lack of user-mode accessible cycle counters and should be easily fixed in future work.

We notice a big divergence in the course of E2E RTT w.r.t. payload size between ICMP and UDP in Figure 6.3: in the two modes that involve FPsPIN for ICMP, the RTT increases almost linearly with the payload size; while in the **Host** mode for ICMP as well as all three modes for UDP, the RTT remains relatively constant. This reflects the difference in checksum calculation be-

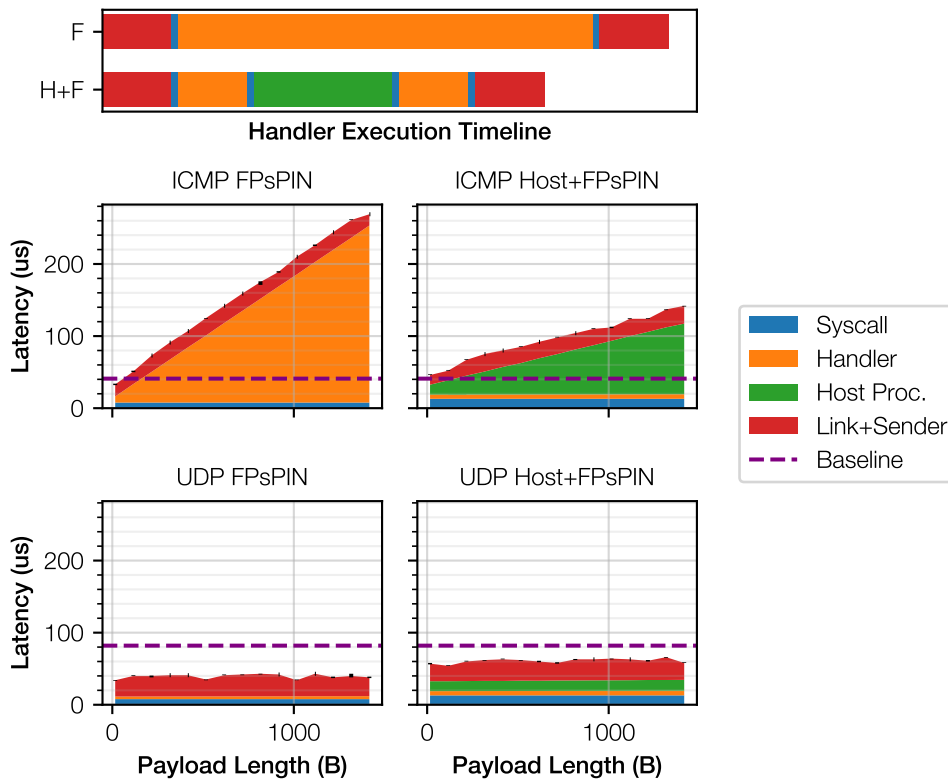


Figure 6.4: Breakdown of the E2E RTT into different categories. The timeline at the top shows the time-series relationship between the various time components (not to scale): *Syscall*, time spent reading the cycles counter from trapping into M-mode; *Handler*, time spent executing the packet handlers, excluding waiting for host; *Host Proc.*, time spent waiting for host DMA and processing on the CPU; and *Sender*, time spent on the Ethernet wire and client side. *Baseline* marks the median E2E RTT in **Host** mode.

tween the ICMP and UDP ping protocols, showing that checksum calculation time is a significant component of the ICMP response RTT. A comparison between the **Host+FPsPIN** and **Host** modes for ICMP in Figure 6.4 reveals that the Linux kernel’s ICMP responder uses extremely optimised code paths, highlighting room of improvement in our IP checksum algorithm.

We further observe that the lower frequency that PULP cores in FPsPIN run at have a significant impact on packet processing latency. This is confirmed by Figure 6.4 between *Handler* on FPsPIN and *Host Proc.* on Host+FPsPIN for ICMP. It shows that a single core on FPsPIN is *only* $\sim 2.8\times$ slower than a single CPU core, a gap way smaller than the actual performance difference between these cores (40 MHz vs 3.4 GHz). Part of this small performance gap comes from the fact that the host CPU performance in checksum calculation is far from ideal due to the CPU always issuing uncached requests to the host DRAM as a result of a lack of cache-coherency over PCIe. In addition, the

host processing category also includes one PCIe RTT between the host CPU and FPsPIN and polling latency, both of which are not present on FPsPIN.

Conclusion The RTT advantage from FPsPIN against the CPU-only **Host** case shows that FPsPIN allows packet processing with lower latency, thanks to its proximity to the data and lack of context switch overheads. The ICMP cases show that FPsPIN still has plenty of potential for higher performance in packet processing from a faster core built for FPGAs (Section 7.1), optimised code that reduces handler execution time, and domain-specific accelerators for compute-heavy workloads like checksum calculation (Section 7.2).

6.3.2 MPI datatypes

Motivation Apart from synthetic benchmarks like the ping-pong demo we showed in the previous section, we also need to demonstrate the ability of FPsPIN to run real-world sPIN workloads. MPI Datatypes [58] is a popular mechanism for exchanging custom messages over the MPI paradigm commonly used in parallel computing. On the sender side, the datatypes subsystem in MPI *serialises* the custom message with non-contiguous memory blocks (in other words, *holes* in between) into a contiguous *streaming* buffer for transmission on the network; on the receiver side, MPI *deserialises* the contiguous message back into the non-contiguous messages for the user application.

Previous work on sPIN has ported the *MPICH dataloop*-based single-threaded implementation of MPI Datatypes to sPIN handlers that run on a simulator-based platform [4]. By porting these existing sPIN handlers to FPsPIN, we characterise the throughput of sPIN workloads on FPsPIN with different levels of handler complexity on the platform. In addition, we showcase the ability of FPsPIN to achieve almost perfect *computation/communication overlap* with a compute-heavy CPU workload that runs simultaneously. Last but not least, since MPI Datatypes requires to send messages of arbitrary length on the network, we demonstrate the operation of the SLMP protocol introduced in Section 5.1.

Experiment We port the handlers in [4] to the FPsPIN platform. A major difference between the original target platform and FPsPIN is the underlying network layer: the handlers were designed for InfiniBand-style networks that offer a *reliable message transport* with arbitrary length support, while FPsPIN runs on top of lossy Ethernet; we have analysed this difference in network-layer guarantees in detail in Section 5.1. For MPI Datatypes, we encapsulate the serialised buffer in SLMP messages on the sender side and send back ACK packets in the handler. In addition to the SLMP encapsulation, we

6.3. Demo Applications on Real Hardware

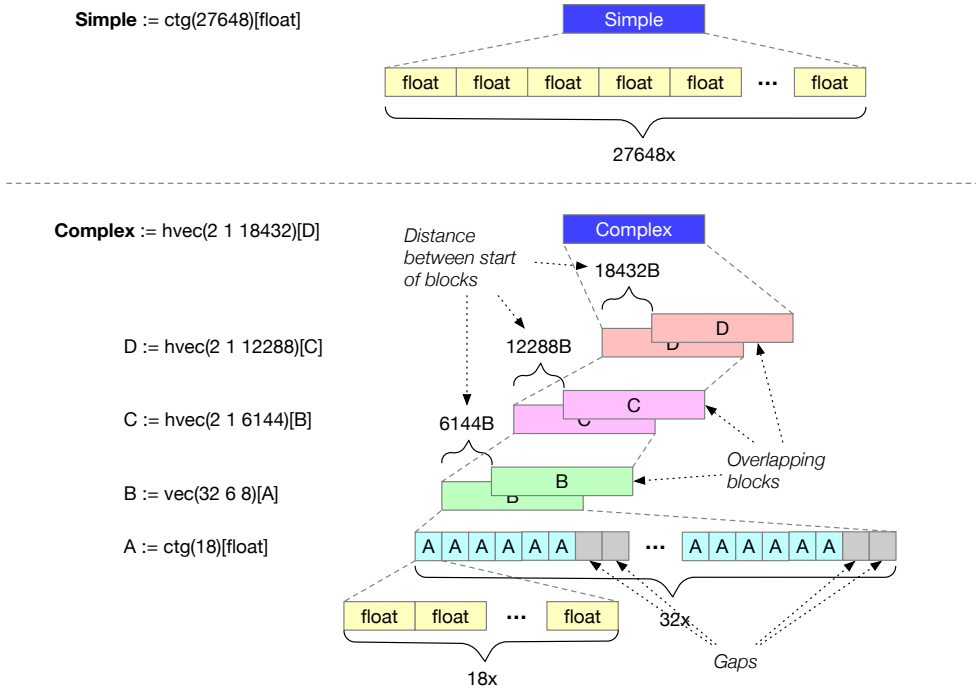


Figure 6.5: Structure of the two datatypes, **Simple** and **Complex**, used for evaluating the datatypes handlers; each layer builds an intermediate nested datatype until we get the final type. ctg: contiguous vector; vec: vector with stride in elements; hvec: vector with stride in bytes.

modify the original handlers to call correct host DMA and host notification functions offered by the FPsPIN runtime.

Due to the single-threaded nature of the *dataloop* implementation of MPI Datatypes, we are unable to implement *packet-level parallelism* and are thus forced to use a sender window size of 1 packet to ensure serialised packet processing; this has severe performance implications as we discussed in Section 5.1. To make use of all the 16 HPUs on FPsPIN, we implement *message-level parallelism*, sending multiple messages in parallel. The handler function stores all per-message states in the shared L2 handler memory and selects the correct per-message state according to the SLMP message ID. We evaluate the E2E bandwidth of two different datatype handlers on FPsPIN in comparison to the reference MPICH datatypes implementation on CPU with varying degrees of parallelism in Figure 6.6. The structures of the two datatype workloads, denoted as **Simple** and **Complex**, are shown in Figure 6.5.

To demonstrate the *computation/communication overlap* capability of FPsPIN, we run double-precision *general matrix multiply* (GEMM) from OpenBLAS [59] on the CPU to simulate a compute-heavy workload that runs simultaneously with the datatypes deserialisation on FPsPIN. Since the CPU fetches notifica-

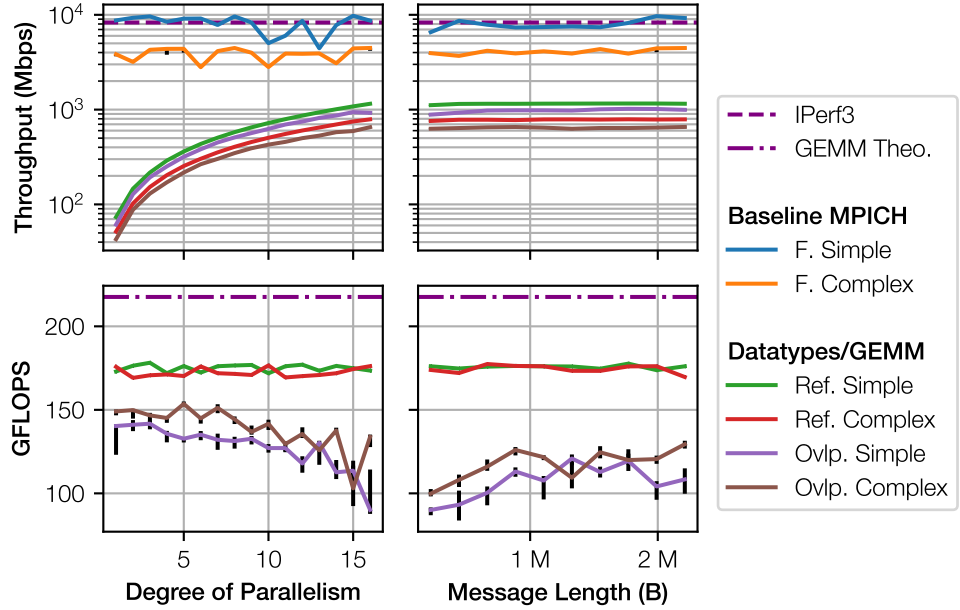


Figure 6.6: Comparison of E2E datatypes and the *general matrix multiply* (GEMM) throughput in diverse parallelism and message length setups. *Ref.* denotes the reference case where the workload is not overlapped with the other; *Ovp.* denotes the overlapped case. We also plot the throughput of the MPICH reference CPU implementation as baseline for comparison. We plot the mean value of 20 measurements in each setup; the error bars in black show the 95% confidence interval.

tions from FPsPIN in poll mode (Section 4.2), it is the best to poll as few times as possible to avoid wasting CPU cycles that could otherwise be running the computation workload. The regularity of *high-performance computing* (HPC) workloads, which allows *estimating* the time a communication task takes to finish. For a meaningful evaluation of computation/communication overlap, we *tune* the GEMM size for a *balanced* setup between computation and communication, i.e. to minimise both datatypes latency and polling overhead. We show in Figure 6.7 the two opposites of polling frequency, both of which hurts the performance of either the datatypes or GEMM. By tuning the size of a single invocation of GEMM, we find the sweet spot to balance between these two situations. Following [60], we define the *overlap ratio* as follows:

$$r_{\text{Overlap}} = \frac{T_{\text{GEMM}}}{T_{\text{GEMM}} + T_{\text{Poll}}}$$

We plot the overlap ratio and polling overhead from two datatypes in Figure 6.8.

In order to have correlated timing measurements between datatypes processing and GEMM, we measure the time elapsed for both workloads on

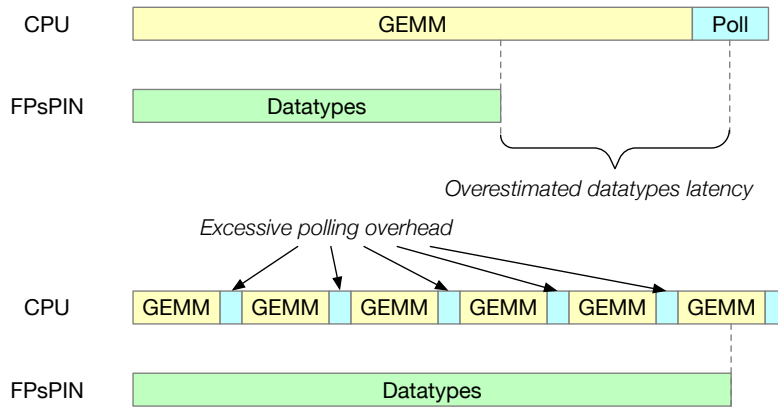


Figure 6.7: Two possible situations of overlap between GEMM and datatypes processing. The top case overestimates the latency of datatypes processing due to not polling at a high-enough frequency; the bottom case imposes excessive overhead on the GEMM workload due to polling too frequently.

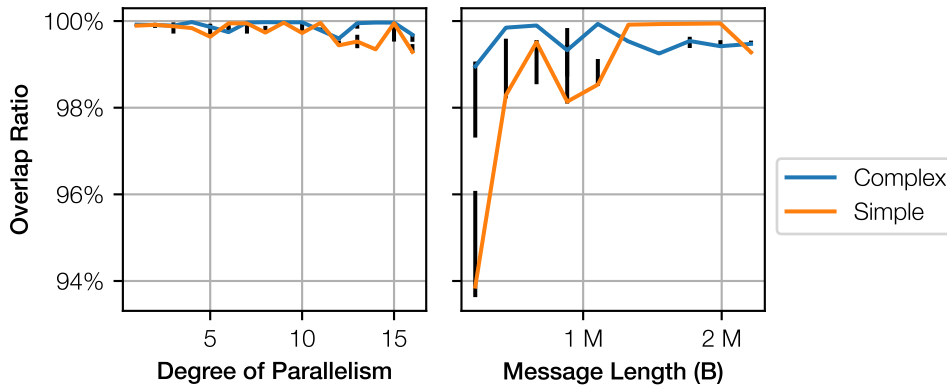


Figure 6.8: Overlap ratio of the two data types as presented in Figure 6.5. The left plot shows the ratio with a fixed message size of 2160 KB across different parallelism settings; the right plot shows the ratio with a fixed parallelism of 16 HPUs across different message sizes. For each configuration, we plot the median values and 95% confidence intervals across 20 samples.

the receiver side. Since the host application does not get a notification until the datatypes transfer is finished, we introduce a *request to send* (RTS) signal from the receiver to the sender: the receiver application sends RTS to the sender and starts the GEMM workload on CPU. We take the time between the notification from FPsPIN and the RTS as the elapsed time for the datatypes workload.

Data interpretation We report the peak throughput of the two datatypes under different message-level parallelism and message length setups as we have shown in Figure 6.6. The highest throughput are achieved at:

- **Simple: 1162.4 Mbps** with message length of 1944 KB and 16 HPUs
- **Complex: 801.2 Mbps** with message length of 1080 KB and 16 HPUs

The throughput gain per extra HPU utilised remains relatively constant at around **73 Mbps** for **Simple** and **50 Mbps** for **Complex** thanks to the *message-level parallelism* mechanism. We believe that this difference comes from the fact that the **Complex** dataloop representation is more complicated and takes more handler time to process compared to **Simple**, manifesting as a lower throughput.

We note in addition that despite the fact that the **Simple** datatype has practically the most simple structure as we have shown in Figure 6.5, the E2E throughput of the datatype is still way lower than the IPerf3 throughput of around 8 Gbps. We conjecture that the MPI dataloop software implementation contributed significantly for a very big overhead in packet handling, resulting in the overall low throughput. We believe that the limitation of having to enforce an SLMP flow control window size of 1 also partly impacted the E2E throughput to some degree. We verify these conjectures in Section 6.3.3 through the comparison of throughput with the synthetic SLMP file transfer benchmark.

We further observe that the GEMM workload suffers from a moderate **20%-30%** slow-down across different parallelism settings when overlapped with datatypes processing, as compared to the reference case. We believe that this slow-down mainly comes from the memory-intensive nature of GEMM, since overlapped datatypes processing would also compete for CPU memory bandwidth through host DMA. We conjecture that the slow-down would be less significant for a more *compute-bound* CPU workload. A similar effect of main memory bandwidth competition can also be observed through a comparison of datatypes processing throughput between the reference and overlapped cases. We recognise that the overlapping ratio stays relatively stable across different degree of parallelism and message sizes.

Figure 6.8 showed a stable overlap ratio of **over 99%** for all parallelism configurations. We also observe that for shorter messages the overlap ratio

drops to as low as 94% due to the short message not allowing longer GEMM workloads within the required number of polls. This is further confirmed by a comparison of the overlap ratio between the **Simple** and **Complex** datatypes, showing that shorter datatypes have lower overlap ratio across all message sizes.

Conclusion We confirm that it is capable to implement and run complicated packet handlers such as the datatypes handler on FPsPIN. The throughput result leaves much to be desired compared to the base throughput from IPerf3, mainly due to the limitation from the MPI dataloop implementation. We evaluate the throughput capability of FPsPIN under normal operating conditions through the synthetic SLMP file transfer benchmark in Section 6.3.3.

Despite the suboptimal throughput results, we demonstrated that FPsPIN allows applications to reliably achieve almost perfect *communication/computation overlap* for sufficiently long messages through overlapping datatypes processing with a synthetic GEMM workload. This successfully shows off the offloading capabilities of FPsPIN.

6.3.3 SLMP file transfer

Motivation As we have seen in the MPI datatypes benchmark in Section 6.3.2, the throughput of packet processing depends highly on the complexity of the packet handlers. It is thus difficult to evaluate the real throughput capacity of FPsPIN with the datatypes handlers due to its various implementation limits. In this demo application, we show the throughput of FPsPIN through a simple file transfer: the sender transmits a file encapsulated in SLMP and the receiver posts the segments to the host CPU for storing. With no requirement of serialised access to per-message states, we evaluate the full potential of *packet-level parallelism* of FPsPIN.

The MPI datatypes demo forced a flow control window size of 1 packet due to the serialisation requirements. However, SLMP as discussed in Section 5.1 offers flexible window size and sender parallelism configurations. We evaluate the performance trade-offs of different sender configurations and identify the best configuration for other applications. In addition, we explore the consequences on packet processing from unreasonable flow control configurations.

Experiment We implement packet handlers that, upon receiving a SLMP segment, DMA the segments to the correct offset in the host memory according to the SLMP message offset field in the packet header. After processing the last packet with the EOM bit, the tail handler posts a notification to the host application, which then saves the host DMA buffer into a file. The sender transmits the file with the SLMP and records the elapsed time for

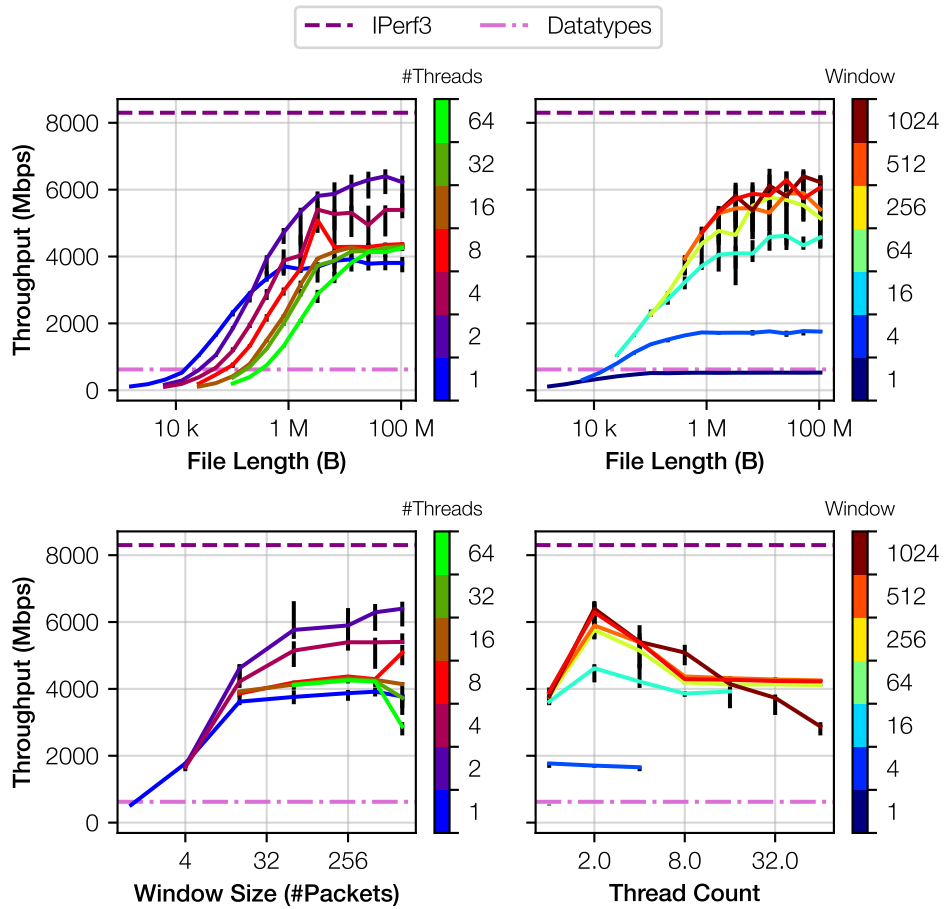


Figure 6.9: Median throughput of both protocols across different sender-side window size and parallelism configurations. The top left plot shows the best throughput across different window sizes; the top right across different thread counts; the bottom two across different file lengths. Each data point comes from 20 successful SLMP message deliveries; the error bars in black show the 95% confidence interval. The *IPerf3* and *Datatypes* lines are from Figure 6.6.

throughput calculation; we use an OpenMP-based multi-threaded version of the `slmp_sendmsg` function offered in `libfspin` (Section 4.2). We report the throughput progressions with diverse file sizes and sender configurations in Figure 6.9.

As we discussed in Section 5.1, a flow control window too big will result in packet loss at the receiver and thus a failure in message delivery. To explore how the flow control settings affect message delivery, we plot the failure rate of message delivery at different sender settings in Figure 6.10. For each configuration, we try to acquire 20 successful runs for a consistent definition of the 95% confidence interval plotted in Figure 6.9; we give up if that takes more than 200 attempts.

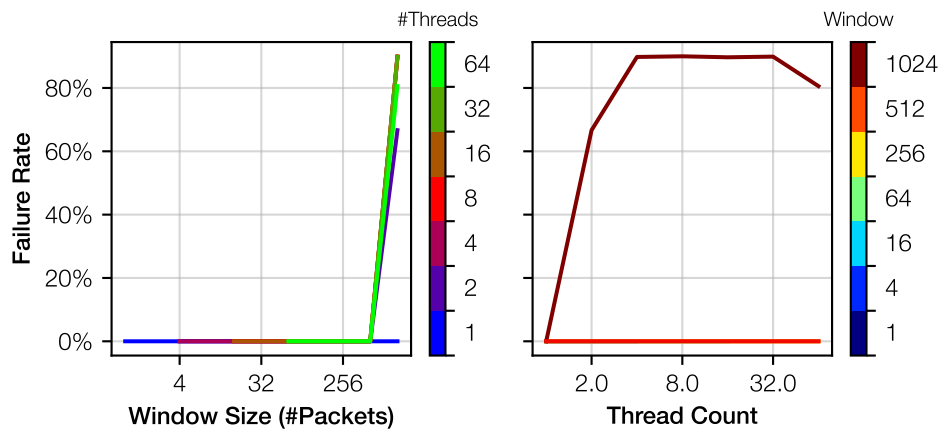


Figure 6.10: Failure rate of SLMP transmissions under different window size and sender thread count configurations. We plot the maximum failure rate acquired with each sender configuration across all file lengths. Note that in the right plot all lines below the window size 1024 overlap with each other at the bottom of the plot.

Data interpretation We observe from Figure 6.9 that the E2E throughput of SLMP file transfers increase along with the length of the file transfer; this is due to the fact that many components in the system require time to setup and spool up in performance, e.g. sender threads and caches, receiver HPUs, etc. The peak *reliable* throughput **6.29 Gbps** is achieved with a sender window size of 512 and 2 threads at a file length of 25 MiB. This is in turn **75.8%** of the mean IPerf3 throughput of 8.3 Gbps. Taking the fact that IPerf3 simply drops the incoming packets without further action, we recognise that this is a reasonable performance number for the SLMP file transfer workload.

We notice that the throughput benefits from a larger sender flow control window size in general. This matches with the theoretical analysis of the flow control window in Section 5.1: a smaller flow control window would force the sender to wait for an ACK sooner than the receive buffer is exhausted, causing the receiver to idle in waiting for packets. In addition, we observe in Figure 6.9 (bottom right) that a larger window needs more than one sender thread to achieve the highest throughput possible. This however does not mean that the more number of sender threads the better, since multiple threads have higher costs to setup and synchronise; the best throughput across most window sizes is achieved with 2 threads.

Although a larger window will result in a good sender bandwidth, this does not always result in a high E2E throughput. We define the *failure rate* of a specific sender configuration as the ratio between runs that finished successfully over all runs sampled. We note from Figure 6.9 (bottom left) that throughput stops increasing beyond a window size of 256 packets; Figure 6.10

SLMP (Mbps)	Simple (Mbps, %)	Complex (Mbps, %)
528	73, 13.8%	50, 9.5%
6290	1162.4, 18%	801.2, 12%

Table 6.4: Comparison of peak throughput between SLMP file transfer and the two datatypes described in Figure 6.5. The percent value shows the ratio of throughput compared to the SLMP application.

shows further that the failure rate skyrockets with a higher window size (1024) and more than one threads (such that the sender can actually saturate the window). As opposed to the peak *reliable* throughput of 6.29 Gbps reported earlier, where the configuration is required to have a 0% failure rate, the achieved highest throughput is **6.4 Gbps** at a 1024-packet window with 45.9% failure rate. This suggests that the actual window size should be somewhere between 512 and 1024.

We compare the throughput results of SLMP file transfer with the datatypes workload introduced in Section 6.3.2. Under different effective degree of parallelism (number of HPUs), the SLMP file transfer shows consistently **one order of magnitude higher** throughput than that of the datatypes, as shown in Table 6.4. Since these two cases have very different SLMP flow control window settings, it is unlikely that flow control played a big role in the low throughput of the datatype handlers. We therefore recognise that the slow-down is largely due to the complicated packet handlers in the datatypes demo taking too long to execute, lowering the throughput.

Conclusion The SLMP file transfer demo application demonstrates the throughput capabilities of FPsPIN under synthetic load, showing throughput close to the IPerf3 baseline case. Comparing that with the MPI Datatypes demo in Section 6.3.2, we conclude that better-optimised handlers that can handle packet-level parallelism would result in higher throughput. Since the slowness of the datatypes demo is largely due to the highly complicated packet handlers, we believe that a more performant FPsPIN with higher F_{\max} will significantly improve the performance for this case. We discuss about such an improvement in Section 7.1.

We evaluated the performance implications from the SLMP protocol introduced in Section 5.1, confirming the importance of setting an accurate flow control window for achieving high E2E throughput. The tedious process of tuning the window size for highest throughput makes a case for protocol-level automatic tuning of the flow control window, as we will discuss in Section 7.3.

Future Work

Don't look at this too closely - you'll go mad. The things we do for performance..

– Linus Torvalds, *linux/arch/alpha/lib/csum_partial_copy.c*

While FPsPIN is relatively complete in terms of functionality, many design choices have been made due to the time constraints of this thesis project. We describe the possible improvements for future work in this chapter.

7.1 Improving F_{\max}

A significant component in the current FPsPIN E2E latency as we have shown in the ICMP and UDP ping-pong demos in Section 6.3.1 is the handler processing latency. This is due to the fact that the PULP cluster was designed for ASIC and not optimised for maximum F_{\max} on FPGAs. We plan to integrate higher frequency RISC-V cores designed for FPGAs, for example VexRiscv¹, as an attempt to achieve higher overall F_{\max} . In addition, we plan to identify and optimise potential critical paths in the FPsPIN hardware components described in Section 6.2; these components had to run at the lower 40 MHz with PULP and thus have not been evaluated for critical path.

Another aspect to explore for a better F_{\max} is the hardware configuration of the PsPIN cluster used in FPsPIN. Although we have made aggressive scale-downs in Table 6.2 compared to the original setup in [7], we did not perform a theoretical analysis of the potential impact to throughput under different workloads for a more informed decision. Proper allocation of L1 and L2 memory and relaxation of the *static* RAM (SRAM) latency requirements would deliver higher performance with lower resource consumption. This

¹<https://github.com/SpinalHDL/VexRiscv>

would in turn reduce congestion that the EDA tools have to handle, allowing a higher F_{\max} .

7.2 Architectural Exploration for HPUs

One of the reasons PULP was chosen as the HPU cores for PsPIN is the PULP DSP instruction set, which contains various custom instructions for bit manipulation and hardware loops. These instructions played an important role in achieving the performance of PsPIN of 200 Gbps. Although it is important to use an HPU core design with a high F_{\max} on FPGAs, the ISA of the core could also vastly affect the packet processing performance and should be chosen carefully: a detailed architectural analysis is needed before investing time into integrating the new core design into the FPsPIN framework.

We have seen significant HPU overhead from IP checksum calculation in the ICMP ping benchmark in Section 6.3.1. This raises the important question of the necessity of *domain-specific accelerators* in a packet processing scenario like sPIN. It is important to balance between flexibility and high-performance: while IP checksum or cryptography accelerators might be useful for almost all packet handlers, a GEMM accelerator might only be used by few applications, leading to a waste in resources. The flexibility of FPsPIN on FPGAs would allow further exploration in finding the right combination of accelerators for high-performance packet processing.

7.3 Advanced Flow Control in SLMP

We discussed in Section 5.1.3 about the importance of flow control to reliable high throughput of application payload. We adopted the approach of exposing the flow control window size to the end user for simplicity in the current implementation. However, as we have shown in Section 6.3.3, it is tedious and error-prone to tune for an accurate window size manually; the more robust approach, as with TCP and QUIC has adopted, is to develop a mechanism to automatically adjust the window size as the protocol operates. Such a mechanism would make the decision to increase or decrease the flow control window based on various metrics such as per-packet RTT, segment loss rate, or explicit receiver buffer state messages. A control law algorithm such as *additive-increase/multiplicative-decrease* (AIMD) from TCP can then be used to actually adjust the window size.

One of the important insight from Portals 4 [61] is that flow control events i.e. receiver buffer overrun should map to an exceptional operation mode of the programming interface, but nevertheless handled correctly and not simply fail. This is also an important insight for sPIN, since the use cases

that sPIN targets could not handle failure well. This prompts the addition of a robust retransmission and error recovery mechanism in SLMP to serve the applications on top of it reliably.

7.4 Parallelise Packet DMA and Scheduling

The current PsPIN two-level scheduler makes the decision of the destination HPU of a packet in one cycle, thanks to the simple round-robin design. However, the scheduler logic will get more complicated to implement future fair scheduling requirements; it will also run at a higher clock frequency, leading to a smaller per-cycle latency budget.

It is thus desirable to parallelise the scheduling decision with the ingress DMA process (discussed in Section 3.2.1) the packet data to hide the scheduling latency. This can be achieved by splitting the HER into two parts. The first part would be issued to the scheduler as soon as the matching engine generates the packet metadata; it contains information necessary for the scheduler to start the scheduling decision (e.g. the ID of the matching ECTX). The second part would be generated after the ingress datapath finished allocating buffer space for the packet buffer and DMA-ing the packet data into the buffer; the scheduler would then actually issue the task to the target HPU. We anticipate that this design would yield the most benefit for a complicated scheduler, in which case the scheduling latency can be hidden in the latency of the ingress datapath.

7.5 Host Notification Queue Pair

The current flag-based host notification mechanism in FPsPIN largely resembles a traditional *queue pair* between the host CPU and the sPIN NIC, with the limitation of only allowing one notification from each HPU at a time. While this limitation is acceptable with the synchronous host notification interface as discussed in Section 4.2, it would not be with asynchronous host notifications since newer notifications would then overwrite older ones that are not yet consumed by the host. It is relatively easy to adapt the current implementation by adding hardware RQ and CQs between the CPU and HPUs. The change would not require changing the software interface, since they already use *push/pop* semantics.

Another potential improvement is to allow *streaming mapping* of the host DMA area. One common use of the host notification interface is to notify the host about a complete message constructed in the host DMA area. However, the host application will be limited in memory access performance to the DMA area due to the current multiple-use DMA mapping. We could implement another ring buffer that hands out *oneshot* buffers for the handlers to DMA

into. The hardware needs to be changed to add the ring buffer and to allow the HER generator to pop and use a new buffer from the ring buffer for every new message; the buffer would be returned to the host to be deallocated later when the HPU sends a host notification.

7.6 More Real-world Applications

In Section 6.3, we showcased the capabilities of FPsPIN through the ICMP and UDP ping-pong, the MPI Datatypes, and the SLMP file transfer demos; we found quite a bit of overlooked points in sPIN, of which we discussed in length in Chapter 5. It would be more beneficial to have more sPIN-enabled workloads ported to FPsPIN to further explore the practicalities of sPIN and the FPGA demo, for example ProtoBuf [5] or distributed filesystems [6]. FPsPIN would also become the starting point for developing new sPIN applications, which would further allow continuous improvement of the system and standard.

7.7 Explore Functionalities from Corundum

Apart from the host DMA subsystem of Corundum that we used in FPsPIN, it offers various other features that would be of interest to explore and possibly integrate into the broader picture of sPIN. One instance is access to memory other than the BRAM or URAM on the FPGA: some devices offer *High Bandwidth Memory* (HBM) or DRAM attached to the FPGA that can be utilised by the NIC. These could potentially be utilised as large scratch areas for the offloaded tasks or extra packet buffer. Corundum also offers extensive support for the *Precision Time Protocol* (PTP), which may enable fine-grain profiling of the internal datapaths in Corundum.

7.8 Stability & Bug Fixes

While the FPsPIN system is complete in function for the demo applications that we implemented and tested in Section 6.3, it is far from bug free. Appendix B.3 lists the current quirks in the system that required special handling during the implementation and evaluation of the demo applications; they should be resolved properly for stable operation of the system in large-scale applications.

Conclusion

“Begin at the beginning,” the King said, very gravely, “and go on till you come to the end: then stop.”

– Lewis Carroll, *Alice’s Adventures in Wonderland*

We set out to build FPsPIN as a faster evaluation platform for sPIN handlers that could run the handlers faster than the cycle-accurate simulation from PsPIN and provide a more complete programming model with host applications. We presented the detailed design and implementation of the hardware (Chapter 3) and software (Chapter 4) components of the system. The successful implementation and evaluation of the ping-pong, MPI datatypes, and SLMP file transfer benchmarks (Chapter 6) shows that this goal has been sufficiently achieved with FPsPIN.

Another important goal of building FPsPIN is to contribute insights and feedback from actually building a sPIN NIC back to the sPIN specification. We identified diverse potential improvements, including reliability protocols, telemetry requirements, scheduler improvements, and many more (Chapter 5). We believe that our feedback would help sPIN become more comprehensive and realistic in achieving its vision for in-network-computing.

Our comprehensive evaluation of FPsPIN through the three benchmarks systematically characterised its performance in latency, throughput, and computation/communication overlap. We showed that FPsPIN achieved stable latency advantage against the baseline host-only case in the ping-pong demo (Section 6.3.1) and near-perfect overlap in the MPI datatypes demo (Section 6.3.2). We also showed that the real-world throughput of FPsPIN in the MPI datatypes benchmark leaves much to be expected compared to the baseline speed test and synthetic SLMP file transfer demo (Section 6.3.3), largely due to the low performance of the measly PULP cores currently used by FPsPIN.

8. CONCLUSION

This shortcoming in throughput provides an outlook of possible improvements to FPsPIN through the integration of a better HPU core designed for FPGAs (Section 7.1). Even more interestingly, it opens up possible future research in HPU architecture on domain-specific acceleration in packet processing (Section 7.2). We are excited to see future progress in this domain.

Appendix A

Reproducing the Results

Alchemy. The link between the immemorial magic arts and modern science. Humankind's first systematic effort to unlock the secrets of matter by reproducible experiment.

– John Ciardi, *Good Words to You*

We present in this appendix how to reproduce the artefacts as well as performance measurements that we have presented in Chapter 6. Most procedures that would take more than one command to finish have a script; we therefore do not go into too much detail explaining the internals of the scripts. The following repositories have been used throughout the project:

- pspin: 9225886259481853dac0a69c20c7b1ac0b74ce36
- corundum: 5351b5e2a66bc8d96b4cc49680baf28916fa1838

Note that corundum checks in pspin as a submodule. The following steps assume that the submodule is set up correctly when cloning the repository.

A.1 Building the System

Hardware To build the FPGA bitstream to be flashed onto the VCU1525 board:

```
1 $ cd corundum/fpga/mqnic/VCU1525/fpga_100g/fpga_pspin/  
2 $ source <path to vivado>/settings64.sh  
3 $ make
```

This will take roughly 6 to 7 hours. The resulting `fpga.bit` is then ready to be flashed onto the FPGA with Vivado. Note that you would need to reboot the host PC which the FPGA board was plugged into to get the BARs correctly recognised and allocated by Linux.

A. REPRODUCING THE RESULTS

You can read the synthesis log or open the resulting project in Vivado to read out the timing and resource utilisation information as presented in Section 6.2.

You will need a DAC cable to connect the two Ethernet ports on the FPGA board to proceed; see Section 6.1 for more explanation.

Software The kernel modules can be built and installed with:

```
1 $ cd corundum/fpga/app/pspin/modules/mqnic
2 $ make && sudo make install
3 $ cd ../mqnic_app_pspin
4 $ make && sudo make install
```

Once they are installed, they will be loaded automatically if the system boots with an appropriate bitstream programmed to the FPGA, or if you rescan the PCIe bus manually:

```
1 $ sudo bash -c 'echo 1 > /sys/bus/pci/<pcie id>/remove'
2 $ sudo bash -c 'echo 1 > /sys/bus/pci/rescan'
```

This is useful if you reprogrammed the FPGA with a new bitstream and need to reload the kernel driver.

The user-space library `libfpspin.a` will be compiled automatically when you compile any of the utilities that depend on it.

Utilities Most of the utilities shipped with FPsPIN are scripts that do not need compiling; however, the Python scripts require setting up a `virtualenv` and installing all the dependencies:

```
1 $ cd $HOME
2 $ python3 -m venv fpspin
3 $ source fpspin/bin/activate
4 $ cd corundum/fpga/app/pspin/utils
5 $ pip install -r requirements.txt
6 $ make # compile the mem.c utility
```

The `mqnic-fw` utility offered by Corundum is useful for programming the *serial peripheral interface* (SPI) configuration flash on the FPGA board, booting the device from the configuration flash, or performing a host reset of the entire FPGA. To build it (and other utilities from Corundum):

```
1 $ cd corundum/utils
2 $ make
```

A.2 Running the Experiments

To compile the FPsPIN handlers, you need the PULP RISC-V toolchain at <https://github.com/pulp-platform/pulp-riscv-gnu-toolchain>. Clone this and follow their instructions to install; after finishing you should have `riscv32-none-elf-gcc` in your `PATH` available.

The four experiments, `icmp_ping`, `udp_ping`, `slmp`, and `datatypes`, follow roughly the same procedure to acquire the data and plot them:

```
1 $ cd corundum/fpga/app/pspin/deps/pspin/examples/<experiment>
2 $ ./run_eval.sh
3 $ ./plot.py
```

The `datatypes` and SLMP benchmark `run_eval.sh` scripts take arguments to resume the experiment from a specific setup if the experiment is interrupted. Read the script to get more information on this.

In the `datatypes` experiment, the machine may sporadically freeze due to quirks mentioned in Appendix B.3 and may require the machine to be reset. You can use the restart functionality built into the run script to restart parts of the experiment.

Appendix B

Debug Facilities

In conclusion, reading and writing a file from within the kernel is a bad, bad thing to do. Never do it. Ever. Both modules from this article, along with a Makefile for compiling them, are available from the Linux Journal FTP site, but we expect to see no downloads in the logs. And, I never told you how to do it either. You picked it up from someone else, who learned it from his sister's best friend, who heard about how to do it from her coworker.

– Greg Kroah-Hartman, *Driving Me Nuts—Things You Never Should Do in the Kernel*

A complex system like FPsPIN requires extensive debugging facilities for diagnosing various issues. We briefly describe the mechanisms we have developed for this purpose.

B.1 Hardware

One important facility to debug the internals of PsPIN is the Verilator cycle-accurate simulation. While this was already available from [7], we implemented a new *interactive* simulation mode that allows the simulation driver to take action when it has received an outgoing packet from PsPIN. This is used to simulate the SLMP flow control window in the datatypes application.

In case a suspected hardware issue is difficult to reproduce in simulation (due to timing or the simulation taking too long), you can use the Xilinx *Integrated Logic Analyzer* (ILA) to inspect signals in the design after it has been implemented and programmed onto the FPGA. Please refer to the Xilinx user manual UG936 [62] for more details.

B.2 Software

A basic facility for debugging handlers on FPsPIN is with the *stdout capture* as we have introduced in Section 4.2. The programmer can simply invoke `printf` in the handler code and then use the `cat_stdout.py` script to read it back on the host. This facility can be used to implement *assertions* to check runtime conditions.

The machine-mode exception handler implemented in the PsPIN runtime will catch exceptions generated in the user space. It will print a message, dump the registers in the state of when the fault happened, and hang the faulting HPU for further investigation. The register dump and program counter allow a programmer to locate the issue by comparing those with a disassembly of the handler's image. It is also possible to read and write to NIC memory locations with the `mem` utility.

The Linux kernel has various facilities to debug a kernel panic, including the `netconsole` that prints the kernel log a.k.a. `dmesg` to a remote host over the network, and `kdump` that would dump the kernel's core memory to disk in the event of a crash. Please refer to the manual of your Linux distribution¹ for more information.

B.3 Quirks & Workarounds

Here is a list of the known issues and quirks in FPsPIN that need attention during operation:

- The `stdout capture` does not have flow control implemented. This means that if multiple HPUs try to write a large amount of data to the FIFO, some characters will be lost when captured on the host.
- The host kernel module, `mqnic_app_pspin.ko`, has a bug that might cause a kernel panic if the driver is detached using `mqnic-fw` while an ECTX is active. Please terminate all running ECTX'es before detaching the driver.
- Due to unidentified bugs in the memory system of PsPIN, handlers running on FPsPIN would occasionally experience memory corruption or HPU hangs, leading to timeouts when polling for notifications in the host application. As a workaround, the developer can reload the ECTX without restarting the host application by calling `fpspin_exit` and then `fpspin_init`.
- The FPsPIN ingress datapath does not correctly handle receive buffer overruns correctly yet; a serious overrun may cause the NIC to stop

¹The user guide from Ubuntu: <https://ubuntu.com/server/docs/kernel-crash-dump>

receiving packets any more, even in bypass mode. Please use flow control whenever possible to avoid this situation; otherwise, please use the `mqnic-fw` utility to reset the whole device and start over.

- The machine may sporadically freeze at a very low level during the datatypes experiment and has to be reset through a power cycle. We suspect that this is due to one or multiple bugs related to low-level PCIe operation in Corundum².
- The design is hard to close timing due to the congestion issues we described in Section 6.2. While the design would still work with a small WNS, issues in PCIe link training might lead to delayed link training and a long time for the device to appear to the host. If link training fails to not complete before the host OS boots, a reboot will be needed to enumerate the device properly.

²Upstream issue: <https://github.com/corundum/corundum/issues/162>

Bibliography

- [1] "Kondi | ASVZ." [Online]. Available: <https://www.asvz.ch/sport/45675-kondi>
- [2] B. Miller, "In Pursuit of 1.6T Data Center Network Speeds | Network Computing." [Online]. Available: <https://www.networkcomputing.com/data-centers/pursuit-16t-data-center-network-speeds>
- [3] T. Hoefler, S. Di Girolamo, K. Taranov, R. E. Grant, and R. Brightwell, "sPIN: High-performance streaming Processing in the Network," *arXiv:1709.05483 [cs]*, Oct. 2017, arXiv: 1709.05483. [Online]. Available: <http://arxiv.org/abs/1709.05483>
- [4] S. Di Girolamo, K. Taranov, A. Kurth, M. Schaffner, T. Schneider, J. Beránek, M. Besta, L. Benini, D. Roweth, and T. Hoefler, "Network-accelerated non-contiguous memory transfers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Denver Colorado: ACM, Nov. 2019, pp. 1–14. [Online]. Available: <https://dl.acm.org/doi/10.1145/3295500.3356189>
- [5] S. Cao, S. Di Girolamo, and T. Hoefler, "Accelerating Data Serialization/Deserialization Protocols with In-Network Compute," in *2022 IEEE/ACM International Workshop on Exascale MPI (ExaMPI)*. Dallas, TX, USA: IEEE, Nov. 2022, pp. 22–30. [Online]. Available: <https://ieeexplore.ieee.org/document/10027020/>
- [6] S. Di Girolamo, D. De Sensi, K. Taranov, M. Malesevic, M. Besta, T. Schneider, S. Kistler, and T. Hoefler, "Building Blocks for Network-Accelerated Distributed File Systems," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. Dallas, TX, USA: IEEE, Nov. 2022, pp. 1–14. [Online]. Available: <https://ieeexplore.ieee.org/document/10046100/>

- [7] S. Di Girolamo, A. Kurth, A. Calotoiu, T. Benz, T. Schneider, J. Beránek, L. Benini, and T. Hoefler, "PsPIN: A high-performance low-power architecture for flexible in-network compute," *arXiv:2010.03536 [cs]*, Jun. 2021, arXiv: 2010.03536. [Online]. Available: <http://arxiv.org/abs/2010.03536>
- [8] A. Forenchich, A. C. Snoeren, G. Porter, and G. Papen, "Corundum: An Open-Source 100-Gbps Nic," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Fayetteville, AR, USA: IEEE, May 2020, pp. 38–46. [Online]. Available: <https://ieeexplore.ieee.org/document/9114811/>
- [9] R. Ross, R. Latham, W. Gropp, E. Lusk, and R. Thakur, "Processing MPI Datatypes Outside MPI," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 5759, pp. 42–53, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-03770-2_11
- [10] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "Offloading distributed applications onto smartNICs using iPipe," in *Proceedings of the ACM Special Interest Group on Data Communication*. Beijing China: ACM, Aug. 2019, pp. 318–333. [Online]. Available: <https://dl.acm.org/doi/10.1145/3341302.3342079>
- [11] X. Wei, R. Cheng, Y. Yang, R. Chen, and H. Chen, "Characterizing Off-path {SmartNIC} for Accelerating Distributed Systems," 2023, pp. 987–1004. [Online]. Available: <https://www.usenix.org/conference/osdi23/presentation/wei-smartnic>
- [12] "Marvell LiquidIO III."
- [13] "Agilio CX SmartNICs - Netronome." [Online]. Available: <https://www.netronome.com/products/agilio-cx/>
- [14] A. Guo, T. Geng, Y. Zhang, P. Haghi, C. Wu, C. Tan, Y. Lin, A. Li, and M. Herbordt, "A Framework for Neural Network Inference on FPGA-Centric SmartNICs," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, Aug. 2022, pp. 01–08, iSSN: 1946-1488.
- [15] Z. Wang, H. Huang, J. Zhang, F. Wu, and G. Alonso, "{FpgaNIC}: An {FPGA-based} Versatile 100Gb {SmartNIC} for {GPUs}," 2022, pp. 967–986. [Online]. Available: <https://www.usenix.org/conference/atc22/presentation/wang-zeke>

-
- [16] N. Corporation, “NVIDIA BlueField-2 DPU - Data Center Infrastructure on a Chip,” 2021. [Online]. Available: <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf>
- [17] 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,” 2018, pp. 51–66. [Online]. Available: <https://www.usenix.org/conference/nsdi18/presentation/firestone>
- [18] Intel, “Intel SmartNICs for Telecommunications at the Broadband Edge.” [Online]. Available: <https://www.intel.com/content/www/us/en/products/programmable/smart-nics-fpga-for-broadband-edge.html>
- [19] Xilinx, “Alveo U25 2x10/25Gb Ethernet PCIe SmartNIC,” 2020. [Online]. Available: <https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/alveo-u25-product-brief.pdf>
- [20] —, “Alveo SN1000 SmartNICs Data Sheet,” 2022.
- [21] M. Khazraee, A. Forenich, G. C. Papen, A. C. Snoeren, and A. Schulman, “Rosebud: Making FPGA-Accelerated Middlebox Development More Pleasant,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. Vancouver BC Canada: ACM, Mar. 2023, pp. 586–605. [Online]. Available: <https://dl.acm.org/doi/10.1145/3582016.3582067>
- [22] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, “P4: programming protocol-independent packet processors,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, Jul. 2014. [Online]. Available: <https://dl.acm.org/doi/10.1145/2656877.2656890>
- [23] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, “Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications,” *ACM Computing Surveys*, vol. 53, no. 1, pp. 1–36, Jan. 2021. [Online]. Available: <https://dl.acm.org/doi/10.1145/3371038>

- [24] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-Programmable Gate Arrays*. Springer Science & Business Media, Jun. 1992.
- [25] A. Limited, "AMBA® AXI Protocol Specification," 2003.
- [26] —, "AMBA AXI-Stream Protocol Specification," 2010.
- [27] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse, and L. Benini, "PULP: A parallel ultra low power platform for next generation IoT applications," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, Aug. 2015, pp. 1–39.
- [28] K. Asanović and D. A. Patterson, "Instruction Sets Should Be Free: The Case For RISC-V," Jun. 2014. [Online]. Available: <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.pdf>
- [29] A. Waterman and K. Asanovic, "The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA," Dec. 2019.
- [30] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine," *Fourth Workshop on Computer Architecture Research with RISC-V*, May 2020.
- [31] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, P. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator," p. 11, Apr. 2016.
- [32] T-HEAD, "T-Head XuanTie C910 High performance RV64 compatible processor." [Online]. Available: <https://img102.alibaba.com/1627958419409/49652c9412c41cb6f39b36fed1244e6e.pdf>
- [33] SiFive, "SiFive Performance™ P650/P670," 2022. [Online]. Available: <https://www.sifive.com/>
- [34] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: an open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*. Heraklion Greece: ACM, Apr. 2020, pp. 1–16. [Online]. Available: <https://dl.acm.org/doi/10.1145/3342195.3387532>
- [35] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, "Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture,"

- in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus OH USA: ACM, Oct. 2019, pp. 14–27. [Online]. Available: <https://dl.acm.org/doi/10.1145/3352460.3358302>
- [36] H. Genc, A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A. Ou, M. Banister, Y. S. Shao, B. Nikolic, I. Stoica, and K. Asanovic, “Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures,” *arXiv:1911.09925 [cs]*, Dec. 2019, arXiv: 1911.09925. [Online]. Available: <http://arxiv.org/abs/1911.09925>
- [37] J. Lin, K. Patel, B. E. Stephens, A. Sivaraman, and A. Akella, “{PANIC}: A {High-Performance} Programmable {NIC} for Multi-tenant Networks,” 2020, pp. 243–259. [Online]. Available: <https://www.usenix.org/conference/osdi20/presentation/lin>
- [38] “Jinja — Jinja Documentation (3.1.x).” [Online]. Available: <https://jinja.palletsprojects.com/en/3.1.x/>
- [39] “Verilog Hierarchical Reference Scope.” [Online]. Available: <https://www.chipverify.com/verilog/verilog-hierarchical-reference-scope>
- [40] D. Cohen and W. Stearns, “IPTables U32 Match Tutorial.” [Online]. Available: <http://www.stearns.org/doc/iptables-u32.current.html>
- [41] W. John and S. Tafvelin, “Analysis of internet backbone traffic and header anomalies observed,” in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. San Diego California USA: ACM, Oct. 2007, pp. 111–116. [Online]. Available: <https://dl.acm.org/doi/10.1145/1298306.1298321>
- [42] T. Benson, A. Anand, A. Akella, and M. Zhang, “Understanding data center traffic characteristics,” in *Proceedings of the 1st ACM workshop on Research on enterprise networking*. Barcelona Spain: ACM, Aug. 2009, pp. 65–72. [Online]. Available: <https://dl.acm.org/doi/10.1145/1592681.1592692>
- [43] D. De Sensi, S. Di Girolamo, S. Ashkboos, S. Li, and T. Hoefler, “Flare: flexible in-network allreduce,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. St. Louis Missouri: ACM, Nov. 2021, pp. 1–16. [Online]. Available: <https://dl.acm.org/doi/10.1145/3458817.3476178>
- [44] A. Forencich, “Verilog PCI Express Components Readme,” Aug. 2023, original-date: 2019-01-08T05:28:51Z. [Online]. Available: <https://github.com/alexforencich/verilog-pcie>

- [45] K. Ichiro, “u-dma-buf(User space mappable DMA Buffer),” Aug. 2023, original-date: 2015-07-13T09:17:35Z. [Online]. Available: <https://github.com/ikwzm/udmabuf>
- [46] “Auxiliary Bus — The Linux Kernel documentation.” [Online]. Available: https://www.kernel.org/doc/html/next/driver-api/auxiliary_bus.html
- [47] P. Mochel and M. Murphy, “sysfs - The filesystem for exporting kernel objects,” Aug. 2011. [Online]. Available: <https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt>
- [48] “IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7,” *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pp. 1–3951, Jan. 2018, conference Name: IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008).
- [49] W. Eddy, “Transmission Control Protocol (TCP),” Internet Engineering Task Force, Request for Comments RFC 9293, Aug. 2022, num Pages: 98. [Online]. Available: <https://datatracker.ietf.org/doc/rfc9293>
- [50] P. J., “User Datagram Protocol,” Internet Engineering Task Force, Request for Comments RFC 768, Aug. 1980, num Pages: 3. [Online]. Available: <https://datatracker.ietf.org/doc/rfc768>
- [51] R. R. Stewart, “Stream Control Transmission Protocol,” Internet Engineering Task Force, Request for Comments RFC 4960, Sep. 2007, num Pages: 152. [Online]. Available: <https://datatracker.ietf.org/doc/rfc4960>
- [52] S. Floyd, M. J. Handley, and E. Kohler, “Datagram Congestion Control Protocol (DCCP),” Internet Engineering Task Force, Request for Comments RFC 4340, Mar. 2006, num Pages: 129. [Online]. Available: <https://datatracker.ietf.org/doc/rfc4340>
- [53] J. Iyengar and M. Thomson, “QUIC: A UDP-Based Multiplexed and Secure Transport,” Internet Engineering Task Force, Request for Comments RFC 9000, May 2021, num Pages: 151. [Online]. Available: <https://datatracker.ietf.org/doc/rfc9000>
- [54] “Incremental Implementation • Vivado Design Suite User Guide: Implementation (UG904) • Reader • AMD Adaptive Computing Documentation Portal.” [Online]. Available: <https://docs.xilinx.com/r/2021.1-English/ug904-vivado-implementation/Incremental-Implementation>

-
- [55] “iputils/iputils,” Aug. 2023, original-date: 2014-04-18T12:14:53Z. [Online]. Available: <https://github.com/iputils/iputils>
- [56] F. Katherine, “stping/dgping: TCP and UDP ping,” Aug. 2023, original-date: 2020-03-30T02:25:36Z. [Online]. Available: <https://github.com/katef/stping>
- [57] T. J. DiCiccio and B. Efron, “Bootstrap confidence intervals,” *Statistical Science*, vol. 11, no. 3, pp. 189–228, Sep. 1996, publisher: Institute of Mathematical Statistics. [Online]. Available: <https://projecteuclid.org/journals/statistical-science/volume-11/issue-3/Bootstrap-confidence-intervals/10.1214/ss/1032280214.full>
- [58] B. C. Pierce, *Types and programming languages*. Cambridge, Mass: MIT Press, 2002.
- [59] Z. Xianyi, W. Qian, and Z. Yunquan, “Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor,” in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, Dec. 2012, pp. 684–691, iSSN: 1521-9097.
- [60] T. Hoefler, A. Lumsdaine, and W. Rehm, “Implementation and performance analysis of non-blocking collective operations for MPI,” in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. Reno Nevada: ACM, Nov. 2007, pp. 1–10. [Online]. Available: <https://dl.acm.org/doi/10.1145/1362622.1362692>
- [61] B. W. Barrett, R. Brightwell, R. E. Grant, W. Schonbein, S. Hemmert, K. Pedretti, K. Underwood, R. Riesen, T. Hoefler, M. Barbe, L. H. S. Filho, A. Ratchov, and A. B. Maccabe, “The Portals 4.3 Network Programming Interface.”
- [62] “Vivado Design Suite User Guide: Creating and Packaging Custom IP (UG1118),” p. 113, 2020.

Acronyms

ACK acknowledgement	HPU handler processing unit
AIMD additive-increase/multiplicative-decrease	ICMP Internet Control Message Protocol
API application programming interface	ILA Integrated Logic Analyzer
ARP Address Resolution Protocol	IMG inter-message gap
ASIC application-specific integrated circuit	IP intellectual property
AXI Advanced eXtensible Interface	IPG inter-packet gap
BAR base address register	IPoIB IP over InfiniBand
BRAM block RAM	ISA instruction set architecture
CDC clock domain crossing	LUT look-up table
CLB configurable logic block	MAC media access control
CMA Contiguous Memory Allocator	MPI Message Passing Interface
CQ completion queue	MPQ message processing queue
DAC direct-attached copper	MTU maximum transmission unit
DCCP Datagram Congestion Control Protocol	NIC network interface card
DMA direct memory access	NISA network instruction set architecture
E2E end-to-end	PBlock placement block
ECTX execution context	PCIe Peripheral Component Interconnect express
EDA electronic design automation	PE processing element
EOM end-of-message	PTP Precision Time Protocol
FF flip-flop	PULP PULP Ultra Low Power
FIFO first-in first-out	QoR quality of results
FPGA field-programmable gate array	RDMA remote direct memory access
GEMM general matrix multiply	RoCE RDMA over Converged Ethernet
HBM High Bandwidth Memory	RQ request queue
HER handler execution request	RTL register transfer level
HPC high-performance computing	RTS request to send
	RTT round-trip time
	SCTP Stream Control Transmission Protocol

ACRONYMS

SDK	software development kit	SYN	synchronisation
SLMP	sPIN Lightweight Messaging Protocol	TCP	Transmission Control Protocol
SoC	system-on-chip	THS	total hold slack
SPI	serial peripheral interface	TNS	total negative slack
sPIN	streaming Processing in the Network	UDP	User Datagram Protocol
SRAM	static RAM	URAM	Ultra RAM
STA	static timing analysis	WHS	worst hold slack
		WNS	worst negative slack



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Full-System Evaluation of the sPIN In-Network-Compute Architecture

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):

Xu

First name(s):

Pengcheng

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zürich, 11.09.2023

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.