hXDP: Efficient Software Packet Processing on FPGA NICs

Anonymous

Abstract

FPGA accelerators on the NIC enable the offloading of expensive packet processing tasks from the CPU. However, FPGAs have limited resources that may need to be shared among diverse applications, and programming them is difficult.

We present a solution to run Linux's eXpress Data Path programs written in eBPF on FPGAs, using only a fraction of the available hardware resources while matching the performance of high-end CPUs. The iterative execution model of eBPF is not a good fit for FPGA accelerators. Nonetheless, we show that many of the instructions of an eBPF program can be compressed, parallelized or completely removed, when targeting a purpose-built FPGA executor, thereby significantly improving performance. We leverage that to design hXDP, which includes (i) an optimizing-compiler that parallelizes and translates eBPF bytecode to an extended eBPF Instruction-set Architecture defined by us; a (ii) soft-CPU to execute such instructions on FPGA; and (iii) an FPGA-based infrastructure to provide XDP's maps and helper functions as defined within the Linux kernel.

We implement hXDP on an FPGA NIC and evaluate it running real-world unmodified eBPF programs. Our implementation is clocked at 156.25MHz, uses about 15% of the FPGA resources, and can run dynamically loaded programs. Despite these modest requirements, it achieves the packet processing throughput of a high-end CPU core and provides a 10x lower packet forwarding latency.

1 Introduction

FPGA-based NICs have recently emerged as a valid option to offload CPUs from packet processing tasks, due to their good performance and re-programmability. Compared to other NIC-based accelerators, such as network processing ASICs [8] or many-core System-on-Chip SmartNICs [35], FPGA NICs provide the additional benefit of supporting diverse accelerators for a wider set of applications [37], thanks to their embedded hardware re-programmability. Notably, Microsoft has been especially advocating for the introduction of FPGA NICs, because of their ability to re-purpose the FPGAs also for tasks such as machine learning [13, 14]. Nonetheless, programming FPGAs is difficult, often requiring the establishment of a dedicated team composed of hardware specialists [18], which interacts with software and operating system developers to integrate the offloading solution with the system.[1]

In this paper, our goal is to provide a more general and easy-to-use solution to program packet processing on FPGA NICs, while seamlessly integrating with existing operating systems. We build towards this goal presenting hXDP, a set of technologies that enables the efficient execution of the Linux's eXpress Data Path (XDP) [24] on FPGA. XDP leverages the eBPF technology to provide secure programmable packet processing within the Linux kernel, and it is widely used by the Linux's community in productive environments. hXDP provides full XDP support, allowing users to dynamically load and run their unmodified XDP programs on the FPGA.

Offloading XDP programs has been already identified as a promising approach to perform packet processing on the NIC, however current solutions only target many-core SoC architectures [35]. In fact, effectively running XDP programs on FPGA is challenging. The eBPF technology used by XDP is originally designed for sequential execution on a high-performance RISC-like register machine. That is, eBPF is designed for server CPUs with high clock frequency and the ability to execute many of the sequential eBPF instructions per second. Instead, FPGAs favor a widely parallel execution model with clock frequencies that are 5-10x lower than those of high-end CPUs. As such, a straightforward implementation of the eBPF iterative execution model on FPGA is likely to provide low packet forwarding performance. Furthermore, unlike other approaches to program packet processing on the FPGA [1, 40, 51], hXDP envisions the sharing of the FPGA resources with other accelerators not related to packet processing tasks. This enables higher levels of consolidation of the infrastructure, which is especially needed by upcoming 5G, local private 5G, and edge computing deployments [11].

[1] While it is in principle possible to use a fixed off-the-shelf FPGA design, using an FPGA for a fixed function partly defeats the motivation for programmable hardware in first place.

Nonetheless, it also adds an additional requirement to the hXDP design, which needs to use little hardware resources to implement arbitrary XDP programs.

We address the challenge performing a detailed analysis of the eBPF Instruction Set Architecture (ISA) and of the existing XDP programs, to reveal and take advantage of opportunities for optimization. First, we identify eBPF instructions that can be safely removed, when not running in the Linux kernel context, by providing targeted hardware support, such as data boundary checks and variable zero-ing. Second, we define extensions to the eBPF ISA to introduce 3-operands instructions, new 6B load/store instructions and a new parametrized program exit instruction. These ISA changes are enabled by the FPGA flexibility, which allows us to implement our own optimized ISA specialized for XDP programs' needs. Finally, we leverage eBPF instruction-level parallelism, performing a static analysis of the programs at compile time, which allows us to execute several eBPF instructions in parallel. We design hXDP to implement these optimizations, and to take full advantage of the on-NIC execution environment, e.g., avoiding unnecessary PCIe transfers. Our design includes: (i) a compiler to translate XDP programs' bytecode to the extended hXDP ISA; (ii) a self-contained FPGA IP Core module that implements the extended ISA alongside several other low-level optimizations; (iii) and the toolchain required to dynamically load and interact with XDP programs running on the FPGA NIC.

To evaluate hXDP we provide an open source implementation for the NetFPGA [54]. We test our implementation using the XDP example programs provided by the Linux source code, and using two real-world applications: a simple stateful firewall; and Facebook's Katran load balancer. hXDP can match the packet forwarding throughput of a multi-GHz server CPU core, while providing a much lower forwarding latency. Compared to a dedicated SoC-based SmartNIC from Netronome, which uses a custom silicon-design with 10s of processing cores clocked at 800MHz, hXDP provides a comparable performance using a single executor clocked at 156MHz. Furthermore, hXDP uses less than 15% of the FPGA resources, making it possible to host multiple different accelerators on the same FPGA. In summary, we contribute:

- the design of hXDP including: the hardware design; the companion compiler; and the software toolchain;
- the implementation of a hXDP IP core for the NetFPGA
- a comprehensive evaluation of hXDP when running real-world use cases, comparing it with an x86 Linux server.

We make our hXDP implementation openly available.

2 Concept and overview

In this section we discuss hXDP goals and requirements, we provide background information about XDP, and finally we present an overview of the hXDP design.

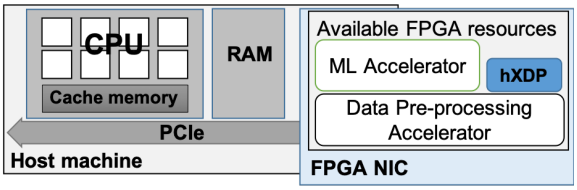

Figure 1: The hXDP concept. hXDP provides an easy-to-use network accelerator that shares the FPGA NIC resources with other application-specific accelerators.

2.1 Goals and Requirements

Goals Our main goal is to provide the ability to run XDP programs efficiently on FPGA NICs, while using little FPGA's hardware resources (See Figure 1).

A little use of the FPGA resources is especially important, since it enables extra consolidation by packing different application-specific accelerators on the same FPGA. This is required, e.g., when the physical space to deploy servers is limited, which is an emerging requirement for smart cities [50], 5G local deployments [39, 42] and for edge computing in general [6, 26].

The choice of supporting XDP is instead motivated by a twofold benefit brought by the technology: it readily enables NIC offloading for already deployed XDP programs; it provides an on-NIC programming model that is already familiar to a large community of Linux programmers. Enabling such a wider access to the technology is important since many of the mentioned edge deployments are not necessarily handled by hyperscale companies. Thus, the companies developing and deploying applications may not have resources to invest in highly specialized and diverse professional teams of developers, while still needing some level of customization to achieve challenging service quality and performance levels.

Non-Goals Unlike other previous work targeting FPGA NICs [1, 40, 51], hXDP does not assume the FPGA to be dedicated to network processing tasks. In fact, hXDP provides dynamic runtime loading of XDP programs, whereas solutions like P4->NetFPGA or FlowBlaze need to often load a new FPGA bitstream when changing application. As such, hXDP does not try to be faster at processing packets than those designs. Instead, hXDP tries to free precious CPU resources while providing similar or better performance than the CPU. Likewise, hXDP does not try to be faster than SmartNICs dedicated to network processing. Such NICs' resources are largely, often exclusively, devoted to network packet processing. Instead, hXDP leverages only a fraction of a FPGA resources to add packet processing with good performance, alongside other application-specific accelerators, which share the same chip's resources.

At the same time, hXDP does not try to be a transparent offloading solution.[2] While the programming model and the

[2]Here, previous complementary work may be applied to help the automatic offloading of network processing tasks [38].

support for XDP are unchanged compared to the Linux implementation, programmers should be aware of which device runs their XDP programs. This is akin to programming for NUMA systems, in which accessing given memory areas may incur additional overheads.

Requirements Given the above discussion, we can derive three high-level requirements for hXDP:

1. it should execute unmodified compiled XDP programs, and support the XDP frameworks' toolchain, e.g., dynamic program loading and userspace access to maps;
2. it should provide packet processing performance at least comparable to that of a high-end CPU core;
3. it should require a small amount of the FPGA's hardware resources.

Before presenting a more detailed description of the hXDP concept, we now give a brief background about XDP.

2.2 XDP Primer

XDP allows programmers to inject programs at the NIC driver level, so that such programs are executed before a network packet is passed to the Linux's network stack. This provides an opportunity to perform custom packet processing at a very early stage of the packet handling, limiting overheads and thus providing high-performance. At the same time, XDP allows programmers to leverage the Linux's kernel, e.g., selecting a subset of packets that should be processed by its network stack, which helps with compatibility and ease of development. XDP is part of the Linux kernel since release 4.18, and it is widely used in production environments [4, 17, 49].

XDP programs are based on the Linux's eBPF technology. eBPF provides an in-kernel virtual machine for the sandboxed execution of small programs within the kernel context. An overview of the eBPF architecture and workflow is provided in Figure 2. In its current version, the eBPF virtual machine has 11 64b registers: $r0$ holds the return value from in-kernel functions and programs, $r1 - r5$ are used to store arguments that are passed to in-kernel functions, $r6 - r9$ are registers that are preserved during function calls and $r10$ stores frame pointer to access the stack. The eBPF virtual machine has a well-defined ISA composed of more than 100 fixed length instructions (64b). The instructions give access to different functional units, such as ALU32, ALU64, branch and memory. Programmers usually write an eBPF program using the C language with some restrictions, which simplify the static verification of the program. Examples of restrictions include forbidden unbound cycles, limited stack size, lack of dynamic memory allocation, etc.

To overcome some of these limitations, eBPF programs can use `helper functions` that implement some common operations, such as checksum computations, and provide access to protected operations, e.g., reading certain kernel memory areas. eBPF programs can also access kernel memory areas called `maps`, i.e., kernel memory locations which essentially resemble tables. Maps are declared and configured at compile

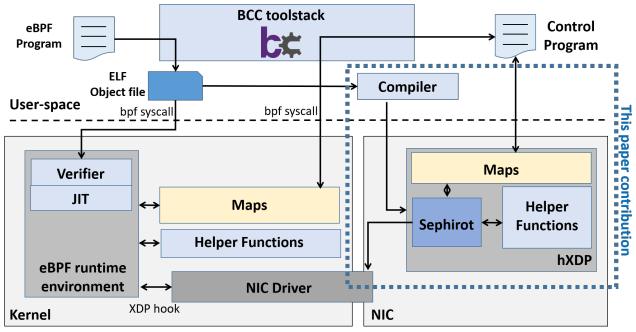

Figure 2: An overview of the XDP workflow and architecture, including the contribution of this paper.

time to implement different data structures, specifying the type, size and an ID. For instance, eBPF programs can use maps to implement arrays and hash tables. An eBPF program can interact with map's locations by means of pointer deference, for un-structured data access, or by invoking specific helper functions for structured data access, e.g., a lookup on a map configured as a hash table.

Maps are especially important since they are the only mean to keep state across program executions, and to share information with other eBPF programs and with programs running in user space. In fact, a map can be accessed using its ID by any other running eBPF program and by the control application running in user space. User space programs can load eBPF programs and read/write maps either using the `libbf` library or frontends such as the BCC toolstack. XDP programs are compiled using LLVM or GCC, and the generated ELF object file is loaded trough the `bpf` syscall, specifying the `XDP hook`. Before the actual loading of a program the kernel verifier checks if it is safe, then the program is attached to the hook, at the network driver level. Whenever the network driver receives a packet, it triggers the execution of the registered programs, which starts from a clean context.

2.3 Challenges

To grasp an intuitive understanding of the design challenge involved in supporting XDP on FPGA, we now consider the example of an XDP program that implements a simple stateful firewall for checking the establishment of bi-directional TCP or UDP flows, and to drop flows initiated from an external location. We will use this function as a running example throughout the paper, since despite its simplicity, it is a realistic and widely deployed function.

The simple firewall first performs a parsing of the Ethernet, IP and Transport protocol headers to extract the flow's 5-tuple (IP addresses, port numbers, protocol). Then, depending on the input port of the packet (i.e., external or internal) it either looks up an entry in a hashmap, or creates it. The hashmap key is created using an absolute ordering of the 5 tuple values, so that the two directions of the flow will map to the same hash. Finally, the function forwards the packet if the input port is internal or if the hashmap lookup retrieved an entry,

otherwise the packet is dropped. A C program describing this simple firewall function is compiled to 71 eBPF instructions.

We can build a rough idea of the potential best-case speed of this function running on an FPGA-based eBPF executor, assuming that each eBPF instruction requires 1 clock cycle to be executed, that clock cycles are not spent for any other operation, and that the FPGA has a 156MHz clock rate, which is common in FPGA NICs [54]. In such a case, a naive FPGA implementation that implements the sequential eBPF executor would provide a maximum throughput of 2.8 Million packets per second (Mpps).[3] Notice that this is a very optimistic upper-bound performance, which does not take into account other, often unavoidable, potential sources of overhead, such as memory access, queue management, etc. For comparison, when running on a single core of a high-end server CPU clocked at 3.7GHz, and including also operating system overhead and the PCIe transfer costs, the XDP simple firewall program achieves a throughput of 7.4 Million packets per second (Mpps). [4] Since it is often undesired or not possible to increase the FPGA clock rate, e.g., due to power constraints, in the lack of other solutions the FPGA-based executor would be 2-3x slower than the CPU core.

2.4 hXDP Overview

hXDP addresses the outlined challenge by taking a software-hardware co-design approach. In particular, hXDP provides both a compiler and the corresponding hardware module. The compiler takes advantage of eBPF ISA optimization opportunities, leveraging hXDP's hardware module features that are introduced to simplify the exploitation of such opportunities. Effectively, we design a new ISA that extends the eBPF ISA, specifically targeting the execution of XDP programs.

The compiler optimizations perform transformations at the eBPF instruction level: remove unnecessary instructions; replace instructions with newly defined more concise instructions; and parallelize instructions execution. All the optimizations are performed at compile-time, moving most of the complexity to the software compiler, thereby reducing the target hardware complexity. We describe the optimizations and the compiler in Section 3. Accordingly, the hXDP hardware module implements an infrastructure to run up to 4 instructions in parallel, implementing a Very Long Instruction Word (VLIW) soft-processor. The VLIW soft-processor does not provide any runtime program optimization, e.g., branch prediction, instruction re-ordering, etc. We rely entirely on the compiler to optimize XDP programs for high-performance execution, thereby freeing the hardware module of complex mechanisms that would use more hardware resources. We describe the hXDP hardware design in Section 4.

Ultimately, the hXDP hardware component is deployed as a self-contained IP core module to the FPGA. The module can be interfaced with other processing modules if needed, or just placed as a bump-in-the-wire between the NIC's port and its PCIe driver towards the host system. The hXDP software toolchain, which includes the compiler, provides all the machinery to use hXDP within a Linux operating system.

From a programmer perspective, a compiled eBPF program could be therefore interchangeably executed in-kernel or on the FPGA, as shown in Figure 2.[5]

```
if (data+sizeof(*eth))>data_end)        r4 = r2
    goto EOP;                           r4 += 14
                                        if r4 > r3 goto +60 <LBB0_17>

struct flow_ctx_table_leaf              r4 = 0
            new_flow = {0};             *(u32 *)(r10-4) = r4
struct flow_ctx_table_key               *(u64 *)(r10-16) = r4
            flow_key = {0};             *(u64 *)(r10-24) = r4
```

Figure 3: Examples of instructions reduced by hXDP

3 hXDP Compiler

In this section we describe the hXDP instruction-level optimizations, and the compiler design to implement them.

3.1 Instructions reduction

The eBPF technology is designed to enable execution within the Linux kernel, for which it requires programs to include a number of extra instructions, which are then checked by the kernel's verifier. When targeting a dedicated eBPF executor implemented on FPGA, most such instructions could be safely removed, or they can be replaced by cheaper embedded hardware checks. Two relevant examples are instructions for memory boundary checks and memory zero-ing.

Boundary checks are required by the eBPF verifier to ensure that programs only read valid memory locations, whenever a pointer operation is involved. For instance, this is relevant for accessing the socket buffer containing the packet data, during parsing. Here, a required check is to verify that the packet is large enough to host the expected packet header. As shown in Figure 3, a single check like this may cost 3 instructions, and it is likely that such checks are repeated multiple times. In the simple firewall case, for instance, there are three such checks for the Ethernet, IP and L4 headers.

Zero-ing is the process of setting a newly created variable to zero, and it is a common operation performed by programmers both for safety and for ensuring correct execution of their programs. A dedicated FPGA executor can provide hard guarantees that all relevant memory areas are zero-ed at program start, therefore making the explicit zero-ing of variables during initialization redundant. In the simple firewall function zero-ing requires 4 instructions, as shown in Figure 3.

3.2 ISA extension

To effectively reduce the number of instructions we define an ISA that enables a more concise description of the program.

[3] I.e., the FPGA can run 156M instructions per second, which divided by the 55 instructions of the program's expected execution path gives a 2.8M program executions per second.

[4] Intel Xeon E5-1630v3, Linux kernel v.5.6.4.

[5] The choice of where to run an XDP program should be explicitly taken by the user, or by an automated control and orchestration system, if available.

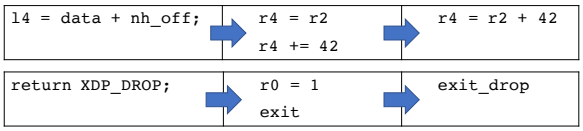

Figure 4: Examples of hXDP ISA extensions

Here, there are two factors at play to our advantage. First, we can extend the ISA without accounting for constraints related to the need to support efficient Just-In-Time compilation. Second, our eBPF programs are part of XDP applications, and as such we can expect packet processing as the main program task. Leveraging these two facts we define a new ISA that changes in three main ways the original eBPF ISA.

Operands number. The first significant change has to deal with the inclusion of three-operands operations, in place of eBPF's two-operands ones. Here, we believe that the eBPF's ISA selection of two operands operations was mainly dictated by the assumption that an x86 ISA would be the final compilation target. Instead, using three operands instructions allows us to replace a two-instructions operation with a single instruction as shown in Figure 4.

Load/store size. The eBPF ISA includes byte-aligned memory load/store operations, with sizes of 1B, 2B, 4B and 8B. While these instructions are effective for most cases, we noticed that during packet processing the use of 6B load/store can reduce the number of instructions in common cases. In fact, 6B is the size of an Ethernet MAC address, which is a commonly accessed field both to check the packet destination or to set a new one. Extending the eBPF ISA with 6B load/store instructions often halves the required instructions.

Parametrized exit. The end of an eBPF program is marked by the exit instruction. In XDP, programs set the $r0$ to a value corresponding to the desired forwarding action (e.g., DROP, TX, etc), then, when a program exits the framework checks the $r0$ register to finally perform the forwarding action (see listing 4). While this extension of the ISA only saves one (runtime) instruction per program, as we will see in Section 4, it will also enable more significant hardware optimizations.

3.3 Instructions Parallelism

Finally, we explore the opportunity to perform parallel processing of an eBPF program's instructions. Here, it is important to notice that high-end *superscalar* CPUs are usually capable to execute multiple instructions in parallel, using a number of complex mechanisms such as speculative execution or out-of-order execution. However, on FPGAs the introduction of such mechanisms could incur significant hardware resources overheads. Therefore, we perform only a static analysis of the instruction-level parallelism of eBPF programs.

To determine if two or more instructions can be parallelized, the three Bernstein conditions have to be checked [3]. Simplifying the discussion to the case of two instructions P_1, P_2:

$$I_1 \cap O_2 = \emptyset; I_1 \cap O_2 = \emptyset; O_2 \cap O_1 = \emptyset; \quad (1)$$

Where I_1, I_2 are the instructions' input sets (e.g. source operands and memory locations) and O_1, O_2 are their output sets. The first two conditions imply that if any of the two instructions depends on the results of the computation of the other, those two instructions cannot be executed in parallel. The last condition implies that if both instructions are storing the results on the same location, again they cannot be parallelized. Verifying the Bernstein conditions and parallelizing instructions requires the design of a suitable compiler, which we describe next.

3.4 Compiler design

We design a custom compiler to implement the optimizations outlined in this section and to transform XDP programs into a schedule of parallel instructions that can run with hXDP. The schedule can be visualized as a virtually infinite set of rows, each with multiple available spots, which need to be filled with instructions. The number of spots corresponds to the number of execution lanes of the target executor. The final objective of the compiler is to fit the given XDP program's instructions in the smallest number of rows. To do so, the compiler performs five steps.

***Control Flow Graph* construction** First, the compiler performs a forward scan of the eBPF bytecode to identify the program's *basic blocks*, i.e., sequences of instructions that are always executed together. The compiler identifies the first and last instructions of a block, and the control flow between blocks, by looking at branching instructions and jumps destinations. With this information it can finally build the *Control Flow Graph* (CFG), which represents the basic blocks as nodes and the control flow as directed edges connecting them.

***Peephole* optimizations** Second, for each basic block the compiler performs the removal of unnecessary instructions (cf. Section 3.1), and the substitution of groups of eBPF instructions with an equivalent instruction of our extended ISA (cf. Section 3.2).

Data Flow dependencies Third, the compiler discovers *Data Flow dependencies*. This is done by implementing an iterative algorithm to analyze the CFG. The algorithm analyzes each block, building a data structure containing the block's input, output, defined, and used symbols. Here, a symbol is any distinct data value defined (and used) by the program. Once each block has its associated set of symbols, the compiler can use the CFG to compute data flow dependencies between instructions. This information is captured in per-instruction *data dependency graphs* (DDG).

Instruction scheduling Fourth, the compiler uses the CFG and the learned DDGs to define an instruction schedule that meets the first two Bernstein conditions. Here, the compiler takes as input the maximum number of parallel instructions the target hardware can execute, and potential hardware constraints it needs to account for. For example, as we will see in Section 4, the hXDP executor has 4 parallel execution lanes, but helper function calls cannot be parallelized.

To build the instructions schedule, the compiler considers one basic block at a time, in their original order in the CFG. For each block, the compiler assigns the instructions to the current schedule's row, starting from the first instruction in the block and then searching for any other *enabled* instruction. An instruction is enabled for a given row when its data dependencies are met, and when the potential hardware constraints are respected. E.g., an instruction that calls a helper function is not enabled for a row that contains another such instruction. If the compiler cannot find any enabled instruction for the current row, it creates a new row. The algorithm continues until all the block's instructions are assigned to a row.

At this point, the compiler uses the CFG to identify potential candidate blocks whose instructions may be added to the schedule being built for the current block. That is, such block's instructions may be used to fill gaps in the current schedule's rows. The compiler considers as candidate blocks the current block's *control equivalent blocks*. I.e, those blocks that are surely going to be executed if the current block is executed. Instructions from such blocks are checked and, if enabled, they are added to the currently existing schedule's rows. This allows the compiler to move in the current block's schedule also a series of branching instructions that are immediately following the current block, enabling a *parallel branching* optimization in hardware (cf. Section 4.2).

When the current block's and its candidate blocks' enabled instructions are all assigned, the algorithm moves to the next block with instructions not yet scheduled, re-applying the above steps. The algorithm terminates once all the instructions have been assigned to the schedule.

Physical register assignment Finally, in the last step the compiler assigns physical registers to the program's symbols. First, the compilers assigns registers that have a precise semantic, such as `r0` for the exit code, `r1`-`r5` for helper function argument passing, and `r10` for the frame pointer. After these fixed assignment, the compiler checks if for every row also the third Bernstein condition is met, otherwise it renames the registers of one of the conflicting instructions, and propagates the renaming on the following dependant instructions.

4 Hardware Module

We design hXDP as an independent IP core, which can be added to a larger FPGA design as needed. Our IP core comprises the elements to execute all the XDP functional blocks on the NIC, including `helper functions` and `maps`. This enables the execution of a program entirely on the FPGA NIC and therefore it avoids as much as possible PCIe transfers.

4.1 Architecture and components

The hXDP hardware design includes five components (see Figure 5): the Programmable Input Queue (`PIQ`); the Active Packet Selector (`APS`); the `Sephirot` processing core; the Helper Functions Module (`HF`); and the Memory Maps Module (`MM`). All the modules work in the same clock frequency

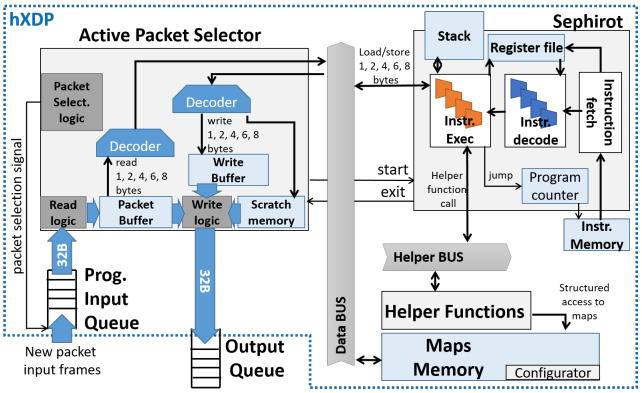

Figure 5: The logic architecture of the hXDP hardware design.

domain. Incoming raw data frames are received by the PIQ. The APS reads a new packet from the PIQ into its internal packet buffer. In doing so, the APS provides a byte-aligned access to the packet data through a *data bus*, which `Sephirot` uses to selectively read/write the packet content. When the APS makes a packet available to the `Sephirot` core, the execution of a loaded eBPF program starts. Instructions are entirely executed within `Sephirot`, using 4 parallel execution lanes, unless they call a helper function or read/write to maps. In such cases, the corresponding modules are accessed using the *helper bus* and the *data bus*, respectively. We detail each components next.

4.1.1 Programmable Input queue

When a packet is received, it enters the Programmable Input Queue (PIQ), which works as an interface with the NIC input bus. Thus, a packet is usually received divided into *frames*, received at each clock cycle. The PIQ holds the packet's frames maintaining a *head frame pointer*. The frames of a given packet can be therefore read from the queue independently from the reception order.

4.1.2 Active Packet Selector

The APS implements a finite-state machine to handle the transfer of a selected packet's frames from the PIQ to an APS' internal buffer.[6] The internal buffer is large enough to hold a full-sized packet.

While the packet is stored divided in frames, the APS provides a byte-aligned read/write access to the data, as required by the eBPF ISA. I.e., the APS implements an eBPF program's *packet buffer*, and builds the hardware-equivalent of the `xdp_md` struct that is passed as argument to XDP programs. `Sephirot` accesses such data structure using the main hXDP's data bus. Since `Sephirot` has four parallel execution lanes, the APS provides four parallel read/write memory accesses through the data bus.

Storing the packet data in frames simplifies the buffer implementation. Nonetheless, this also makes the writing of

[6]The policy for selecting a given packet from the PIQ is by default FIFO, although this can be changed to implement more complex mechanisms.

specific bytes in the packet more complex. In particular, since only a frame-size number of bytes can be written to the buffer, the writing of single bytes would need to first read the entire frame, apply the single-byte modification, and then re-write to the buffer the entire modified frame. This is a complex operation, which would impact the maximum achievable clock rate if implemented in this way, or it would alternatively require multiple clock cycles to be completed. We instead use a *difference buffer* to handle writes, trading off some memory space for hardware complexity. That is, modifications to the packet data are stored in a difference buffer that is byte addressed. As we will see next, the difference buffer allows us to separate the reading of certain packet data, which can be pre-fetched by `Sephirot` during the decoding of an instruction, from the actual writing of new data in the packet, which can be performed during packet emission. In fact, the APS contains also a scratch memory to handle modifications to the packet that are applied before the current packet head. This is usually required by applications that use the `bpf_adjust_head` helper.

The scratch memory, the difference buffer, and the packet buffer are combined during packet emission to output the modified packet data. The entire process is handled by a dedicated finite-state machine, which is started by `Sephirot` when an *exit* instruction is executed. The emission of a packet happens in parallel with the reading of the next packet.

4.1.3 `Sephirot`

`Sephirot` is a VLIW processor with 4 parallel lanes that execute eBPF instructions. `Sephirot` is designed as a pipeline of four stages: instruction fetch (IF); instruction decode (ID); instruction execute (IE); and commit. A program is stored in a dedicated instruction memory, from which `Sephirot` fetches the instructions in order. The processor has another dedicated memory area to implement the program's stack, which is 512B in size, and 11 64b registers stored in the register file. These memory and register locations match one-to-one the eBPF virtual machine specification. `Sephirot` begins execution when the APS has a new packet ready for processing, and it gives the processor *start* signal.

On processor start (IF stage) a VLIW instruction is read and the 4 extended eBPF instructions that compose it are statically assigned to their respective execution lanes. In this stage, the operands of the instructions are pre-fetched from the register file. The remaining 3 pipeline stages are performed in parallel by the four execution lanes. During ID, memory locations are pre-fetched, if any of the eBPF instructions is a *load*, while at the IE stage the relevant sub-unit are activated, using the relevant pre-fetched values. The sub-units are the Arithmetic and Logic Unit (ALU), the Memory Access Unit and the Control Unit. The ALU implements all the operations described by the eBPF ISA, with the notable difference that it is capable of performing operations on three operands. The memory access unit abstracts the access to the different memory areas, i.e., the stack, the packet data stored in the APS, and the maps memory. The control unit provides the logic to modify the program counter, e.g., to perform a *jump*, and to invoke helper functions. Finally, during the commit stage the results of the IE phase are stored back to the register file, or to one of the memory areas. `Sephirot` terminates execution when it finds an exit instruction, in which case it signals to the APS the packet forwarding decision.

4.1.4 Helper Functions

hXDP implements the XDP helper functions in a dedicated sub-module. We decided to provide a dedicated hardware implementation for the helper functions since their definition is rather static, and it changes seldom when new versions of the XDP technology are released. This also allows us to leverage at full the FPGA hardware parallelism to implement some more expensive functions, such as checksum computations. In terms of interface, the helper function sub-module offers the same interface provided by eBPF, i.e., helper functions arguments are read from registers r1-r5, and the return value is provided in r0. All values are exchanged using the dedicated helper data bus. Here, it is worth noticing that there is a single helper functions sub-module, and as such only one instruction per cycle can invoke a helper function.[7] Among the helper functions there are the map lookup functions, which are used to implement hashmap and other data structures on top of the maps memory. Because of that, the helper funcions sub-module has a direct access to the maps module.

4.1.5 Maps

The maps subsystem main function is to decode memory addresses, i.e., map id and row, to access the corresponding map's memory location. Here, one complication is that eBPF maps can be freely specified by a program, which defines the map's type and size for as many maps as needed. To replicate this feature in the hardware, the maps subsystem implements a *configurator* which is instructed at program's load time. In fact, all the maps share the same FPGA memory area, which is then shaped by the configurator according to the maps section of the eBPF program, which (virtually) creates the appropriate number of maps with their row sizes, width and hash functions, e.g., for implementing hashmaps.

Since in eBPF single maps entries can be accessed directly using their address, the maps subsystem is connected via the data bus to `Sephirot`, in addition to the direct connection to the helper function sub-module, which is instead used for structured map access. To enable parallel access to the `Sephirot`'s execution lanes, like in the case of the APS, the maps modules provides up to 4 read/write parallel accesses.

[7] Adding more sub-modules would not be sufficient to improve parallelism in this case, since we would need to also define additional registers to hold arguments/return values and include register renaming schemes. Adding sub-modules proved to be not helpful for most use cases.

Figure 6: Example of a switch statement

4.2 Pipeline Optimizations

Early processor start The packet content is transferred one frame per clock cycle from the PIQ to the APS. Starting program execution without waiting the full transfer of the packet may trigger the reading of parts of it that are not yet transferred. However, handling such an exception requires only little additional logic to pause the `Sephirot` pipeline, when the exception happens. In practice, XDP programs usually start reading the beginning of a packet, in fact in our tests we never experienced a case in which we had to pause `Sephirot`. This provides significant benefits with packets of larger sizes, effectively masking the `Sephirot` execution time.

Program state self-reset As we have seen in Section 3, eBPF programs may perform zero-ing of the variables they are going to use. We provide automatic reset of the stack and of the registers at program initialization. This is an inexpensive feature in hardware, which improves security [15] and allows us to remove any such zero-ing instruction from the program.

Data hazards reduction One of the issues of pipelined execution is that two instructions executed back-to-back may cause a race condition. If the first instruction produces a result needed by the second one, the value read by the second instruction will be stale, because of the operand/memory pretecthing performed by `Sephirot`. Stalling the pipeline would avoid such race conditions at the cost of performance. Instead, we perform result forwarding on a per-lane basis. This allows the scheduling back-to-back of instructions on a single lane, even if the result of the first instruction is needed by the second one. The compiler is in charge of checking such cases and ensure that the instructions that have such dependancies are always scheduled on the same lane.

Parallel branching The presence of branch instructions may cause performance problems with architectures that lack branch prediction, speculative and out of order execution. In the case of `Sephirot`, this forces a serialization of the branch instructions However, in XDP programs there are often series of branches in close sequence, especially during header parsing (see Figure 6). We enabled the parallel execution of such branches, establishing a priority ordering of the `Sephirot`'s lanes. The compiler takes that into account when scheduling instructions, ordering the branch instructions accordingly.[8]

Early processor exit The processor stops when an exit instruction is executed. The exit instruction is recognized during the IF phase, which allows us to stop the processor pipeline early, and save the 3 remaining clock cycles. This optimization improves also the performance gain obtained by extending the ISA with parametrized exit instructions, as described in Section 3. In fact, XDP programs usually perform a move of a value to $r0$, to define the forwarding action, before calling an exit. Setting a value to a register always needs to traverse the entire `Sephirot` pipeline. Instead, with a parametrized exit we remove the need to assign a value to $r0$, since the value is embedded in a newly defined exit instruction.

[8] This applies equally to a sequence of `if...else` or `goto` statements.

Table 1: NetFPGA resources usage breakdown, each row reports actual number and percentage of the FPGA total resources (#, % tot). hXDP requires about 15% of the FPGA resources in terms of Slice Logic and Registers.

Component	Logic	Registers	BRAM
PIQ	215, 0.05%	58, <0.01%	6.5, 0.44%
APS	9K, 2.09%	10K, 1.24%	4, 0.27%
Sephirot	27K, 6.35%	4K, 0,51%	-
Instr Mem	-	-	7.7, 0.51%
Stack	1K, 0.24%	136, 0.02%	16, 1,09%
HF Subsystem	339, 0.08%	150, 0.02%	-
Maps Subsystem	5.8K, 1.35%	2.5K, 0.3%	16, 1.09%
total	42K, 9.91%	18K, 2.09%	50, 3.40%
w/ reference NIC	80K, 18.53%	63K, 7.3%	214, 14,63%

4.3 Implementation

We prototyped hXDP using the NetFPGA [54], a board embedding 4 10Gb ports and a Xilinx Virtex7 FPGA. The hXDP implementation uses a frame size of 32B and is clocked at 156.25MHz. Both settings come from the standard configuration of the NetFPGA reference NIC design.

The hXDP FPGA IP core takes 9.91% of the FPGA logic resources, 2.09% of the register resources and 3.4% of the FPGA's available BRAM. The considered BRAM memory does not account for the variable amount of memory required to implement maps. A per-component breakdown of the required resources is presented in Table 1, where for reference we show also the resources needed to implement a map with 64 rows of 64B each. As expected, the APS and `Sephirot` are the components that need more logic resources, since they are the most complex ones. Interestingly, even somewhat complex helper functions, e.g., a helper function to implement a hashmap lookup (`HF Map Access`), have just a minor contribution in terms of required logic, which confirms that including them in the hardware design comes at little cost while providing good performance benefits, as we will see in Section 5. When including the NetFPGA's reference NIC design, i.e., to build a fully functional FPGA-based NIC, the overall occupation of resources grows to 18.53%, 7.3% and 14.63% for logic, registers and BRAM, respectively. This is a relatively low occupation level, which enables the use of the largest share of the FPGA for other accelerators.

Program	Description
xdp1	parse pkt headers up to IP, and XDP_DROP
xdp2	parse pkt headers up to IP, and XDP_TX
xdp_adjust_tail	receive pkt, modify pkt into ICMP pkt and XDP_TX
router_ipv4	parse pkt headers up to IP, look up in routing table and forward (redirect)
rxq_info (drop)	increment counter and XDP_DROP
rxq_info (tx)	increment counter and XDP_TX
tx_ip_tunnel	parse pkt up to L4, encapsulate and XDP_TX
redirect_map	output pkt from a specified interface (redirect)

Table 2: Tested Linux XDP example programs.

5 Evaluation

We use a selection of the Linux's XDP example applications and two real world applications to perform the hXDP evaluation. The Linux examples are described in Table 2. The real-world applications are the simple firewall we used as running example, and the Facebook's Katran server load balancer [17]. Katran is a high performance software load balancer that translates virtual addresses to actual server addresses using a weighted scheduling policy, and providing per-flow consistency. Furthermore, Katran collects several flow metrics, and performs IPinIP packet encapsulation.

Using these applications, we perform an evaluation of the impact of the compiler optimizations on the programs' number of instructions, and the achieved level of parallelism. Then, we evaluate the performance of our NetFPGA implementation. In addition, we run a large set of micro-benchmarks to highlight features and limitations of hXDP.

5.1 Compiler

Instruction-level optimizations We evaluate the instruction-level optimizations described in Section 3, by activating selectively each of them in the hXDP compiler. Figure 7 shows the reduction of eBPF instructions for a program, relative to its original number of instructions. We can observe that the contribution of each optimization largely depends on the program. For instance, the `xdp_adjust_tail` performs several operations that need to read/write 6B of data, which in turn makes the 6B load/store instructions of our extended ISA particularly effective, providing a 18% reduction in the number of instructions. Likewise, the `simple_firewall` performs several bound checks, which account for 19% of the program's instructions. The parametrized exit reduces the number of instructions by up to 5-10%. However, it should be noted that this reduction has limited impact on the number of instructions executed at runtime, since only one exit instruction is actually executed.

Instructions parallelization We configure the compiler to consider from 2 to 8 parallel execution lanes, and count the number of generated VLIW instructions. A VLIW instruction corresponds to a schedule's row (cf. Section 3.4), and it can therefore contain from 2 to 8 eBPF instructions in this test. Figure 8 shows[9] that the number of VLIW instructions is reduced significantly as we add parallel execution lanes up to 3, in all the cases. Adding a fourth execution lane reduces the VLIW instructions by an additional 5%, and additional lanes provide only marginal benefits. Another important observation is that the compiler's physical register assignment step becomes more complex when growing the number of lanes, since there may not be enough registers to hold all the symbols being processed by a larger number of instructions.[10] Given the relatively low gain when growing to more than four parallel lanes, we decided use four parallel lanes in hXDP.

Combined optimizations Figure 9 shows the final number of VLIW instructions produced by the compiler. We show the reduction provided by each optimization as a stacked column, and report also the number of x86 instructions, which result as output of the Linux's eBPF JIT compiler. In Figure, we report the gain for instruction parallelization, and the additional gain from *code movement*, which is the gain obtained by anticipating instructions from control equivalent blocks (cf. Section 3.4). As we can see, when combined, the optimizations do not provide a simple sum of their gains, since each optimization affects also the instructions touched by the other optimizations. Overall, the compiler is capable of providing a number of VLIW instructions that is often 2-3x smaller than the original program's number of instructions. Notice that, by contrast, the output of the JIT compiler for x86 usually grows the number of instructions.[11]

5.2 Hardware performance

We compare hXDP with XDP running on a server machine, equipped with an Intel Xeon E5-1630 v3 @3.70GHz, an Intel XL710 40GbE NIC, and running Linux v.5.6.4 with the i40e Intel NIC drivers. During the tests we use different CPU frequencies, i.e., 1.2GHz, 2.1GHz and 3.7GHz, to cover a larger spectrum of deployment scenarios. In fact, many deployments favor CPUs with lower frequencies and a higher number of cores [22]. We use a DPDK packet generator to perform throughput and latency measurements. The packet generator is capable of generating a 40Gbps throughput with any packet size and it is connected back-to-back with the system-under-test, i.e., the hXDP prototype running on the NetFPGA or the Linux server running XDP. Delay measurements are performed using hardware packet timestamping at the traffic generator's NIC, and measure the round-trip time. Unless differently stated, all the tests are performed using packets with size 64B, which are the most challenging workload for both systems under test. Since we are interested in measuring the hXDP hardware implementation performance, we do not perform tests that require moving packets to the host system. In such cases the packet processing performance would be largely affected by the PCIe and Linux drivers implementations, which are out-of-scope for this paper.

[9] For readability we only show up to 6 lanes.

[10] This would ultimately require adding more registers, or the introduction of instructions to handle register spilling.

[11] This is also due to the overhead of running on a shared executor, e.g., calling helper functions requires several instructions.

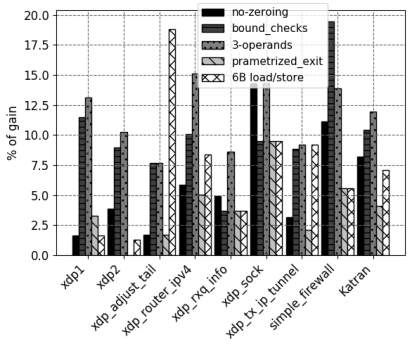

Figure 7: Reduction of instructions due to compiler optimizations, relative to the original number of instructions.

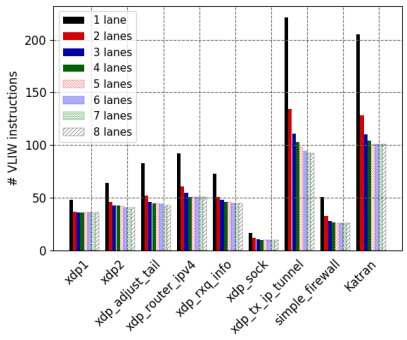

Figure 8: Number of VLIW instructions when varying the available number of execution lanes.

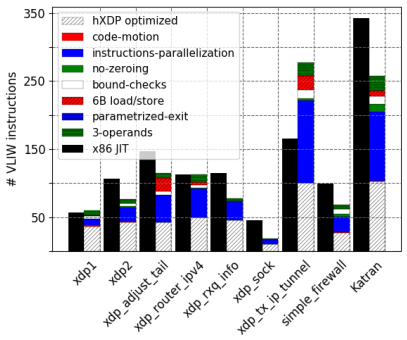

Figure 9: Number of VLIW instructions, and impact of optimizations on its reduction.

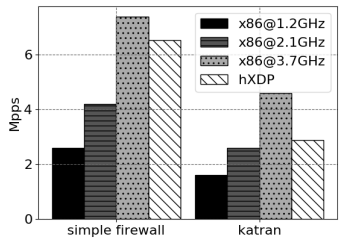

Figure 10: Throughput for real-world applications. hXDP is faster than a high-end CPU core clocked at over 2GHz.

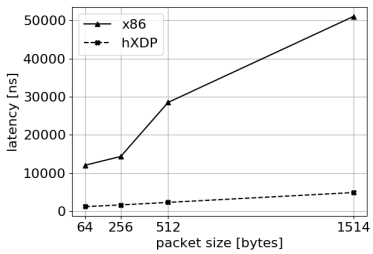

Figure 11: Packet processing latency when running the simple firewall program, for different packet sizes.

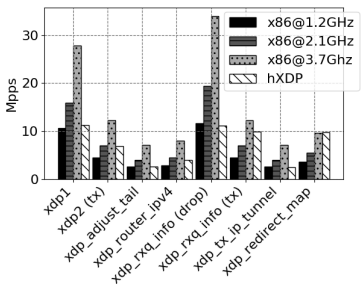

Figure 12: Throughput of Linux's XDP programs. hXDP is faster for programs that perform TX or redirection.

5.2.1 Applications performance

Simple firewall In Section 2 we mentioned that an optimistic upper-bound for the hardware performance would have been 2.8Mpps. When using hXDP with all the compiler and hardware optimizations described in this paper, the same application achieves a throughput of 6.53Mpps, as shown in Figure 10. This is only 12% slower than the same application running on a powerful x86 CPU core clocked at 3.7GHz, and 55% faster than the same CPU core clocked at 2.1GHz. In terms of latency, hXDP provides about 10x lower packet processing latency, for all packet sizes (see Figure 11). We omit latency results for the remaining applications, since they are not significantly different.[12]

Katran When measuring Katran we find that hXDP is instead 38% slower than the x86 core at 3.7GHz, and only 8% faster than the same core clocked at 2.1GHz. The reason for this relatively worse hXDP performance is the overall program length. Katran's program has many instructions, as such executors with a very high clock frequency are advantaged, since they can run more instructions per second. However, notice the clock frequencies of the CPUs deployed at Facebook's datacenters [22] have frequencies close to 2.1GHz, favoring many-core deployments in place of high-frequency ones. hXDP clocked at 156MHz is still capable of outperforming a CPU core clocked at that frequency.

Linux examples We finally measure the performance of the Linux's XDP examples listed in Table 2. These applications allow us to better understand the hXDP performance with programs of different types (see Figure 12). We can identify three categories of programs. First, programs that forward packets to the NIC interfaces are faster when running on hXDP. These programs do not pass packets to the host system, and thus they can live entirely in the NIC. For such programs, hXDP usually performs at least as good as a single x86 core clocked at 2.1GHz. In fact, processing XDP on the host system incurs the additional PCIe transfer overhead to send the packet back to the NIC. Second, programs that always drop packets are usually faster on x86, unless the processor has a low frequency, such as 1.2GHz. Here, it should be noted that such programs are rather uncommon, e.g., programs used to gather network traffic statistics receiving packets from a network tap. Finally, programs that are long, e.g., `tx_ip_tunnel` has 283 instructions, are faster on x86. Like we noticed in the case of Katran, with longer programs the hXDP's implementation low clock frequency can become problematic.

[12] The impact of different programs is especially significant with small packet sizes. However, even in such cases we cannot observe significant differences. In fact each VLIW instruction takes about 7 nanoseconds to be executed, thus, differences of tens of instructions among programs change the processing latency by far less than a microsecond.

5.2.2 Microbenchmarks

Baseline We measure the baseline packet processing performance using three simple programs: *XDP_DROP* drops the packet as soon as it is received; *XDP_TX* parses the Ethernet header and swaps MAC addresses before sending the packet out to the port from which it was received; *redirect* is like XDP_TX, but sends the packet out to a different port, which requires calling a specific XDP helper function. The performance results clearly show the advantage of running on the NIC and avoiding PCIe transfers when processing small programs (see Figure 13). hXDP can drop 52Mpps vs the 38Mpps of the x86 CPU core@3.7GHz. Here, the very high performance of hXDP is due to the parametrized exit/early exit optimizations mentioned in Section 4. Disabling the optimization brings down the hXDP performance to 22Mpps. In the case of XDP_TX, instead, hXDP forwards 22.5Mpps while x86 can forward 12Mpps. In the case of redirect, hXDP provides 15Mpps, while x86 can only forward 11Mpps when running at 3.7GHz. Here, the redirect has lower performance because eBPF implements it with a helper.

Maps access Accessing eBPF maps affects the performance of XDP programs that need to read and keep state. In this test we measure the performance variation when accessing a map with a variable key size ranging between 1-16B. Accessing the map is performed calling a helper function that performs a hash of the key and then retrieves the value from memory. In the x86 tests we ensure that the accessed entry is in the CPU cache. Figure 14 shows that hXDP prototype has constant access performance, independently from the key size. This is the result of the wider memory data buses, which can in fact accomodate a memory access in a single clock cycle for keys of up to 32B in size. Instead, in the x86 case the performance drops when the key size grows from 8B to 16B, we believe this is due to the need to access multiple cache entries.

Helper functions In this micro-benchmark we measure throughput performance when calling from 1 to 40 times a helper function that performs an incremental checksum calculation. Since helper functions are implemented as dedicated hardware functions in hXDP, we expect our prototype to exhibit better performance than x86, which is confirmed by our results (see Figure 15). I.e., hXDP may provide significant benefits when offloading programs that need complex computations captured in helper functions. Also, the hXDP's helper function machinery may be used to eventually replace sets of common instructions with more efficient dedicated hardware implementations, providing an easier pathway for future extensions of the hardware implementation.

Instruction per cycle We compare the parallelization level obtained at compile time by hXDP, with the runtime parallelization performed by the x86 CPU core. Table 3 shows that despite lacking features such as speculative execution and branch prediction, the static hXDP parallelization achieves often a parallelization level as good as the one achieved by the complex x86 runtime machinery.[13]

Program	# instr.	x86 IPC	hXDP IPC
xdp1	61	2.20	1.70
xdp2	78	2.19	1.81
xdp_adjust_tail	117	2.37	2.72
router_ipv4	119	2.38	2.38
rxq_info	81	2.81	1.76
tx_ip_tunnel	283	2.24	2.83
simple_firewall	72	2.16	2.66
Katran	268	2.32	2.62

Table 3: Programs' number of instructions, x86 runtime instruction-per-cycle (IPC) and hXDP static IPC mean rates.

6 Discussion and opportunities

Suitable applications hXDP can run XDP programs with no modifications, however, the results of Section 5 show that hXDP is especially suitable for programs that can process packets entirely on the NIC, and which are no more than a few 10s of VLIW instructions long. This is a common observation made also for other offloading solutions [23].

FPGA Sharing At the same time, hXDP succeeds in using little FPGA resources, leaving space for other accelerators. For instance, we could co-locate on the same FPGA several instances of the VLDA accelerator design for neural networks presented in [12]. Here, one important note is about the use of memory resources (BRAM). Some XDP programs may need larger map mameories. It should be clear that the variable memory area dedicated to maps is also shared with other accelerators on the FPGA. As such, considering memory requirements of XDP programs, which are anyway known at compile time, is another important factor to consider when taking program offloading decisions.

Future work While the hXDP performance results are already good to run real-world applications, e.g., Katran, we identified a number of optimization options, as well as avenues for future research. First, our compiler can be improved. For instance, we were able to hand-optimize the simple firewall instructions and run it at 7.1Mpps on hXDP. The applied optimizations had to do with a better organization of the memory accesses, and we believe they could be automated. Second, XDP programs often have large sections dedicated to packet parsing. Identifying them and providing in hardware a dedicated programmable parser [21] may significantly reduce the number of instructions executed by hXDP. Third, while we focused on a single processing core in this paper, hXDP can be extended to support two or more `Sephirot` cores. This would effectively trade off more FPGA resources for higher forwarding performance. For instance, we could test an implementation with two cores, and two lanes each, with little effort. This was the case since the two cores shared a common

[13] The x86 IPC should be understood as a coarse-grained estimation of the XDP instruction-level parallelism since, despite being isolated, the CPU runs also the operating system services related to the eBPF virtual machine, and its IPC is also affected by memory access latencies, which more significantly impact the IPC for high clock frequencies.

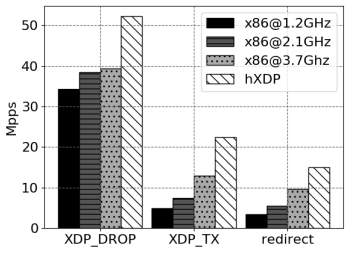

Figure 13: Baseline throughput measurements for basic XDP programs.

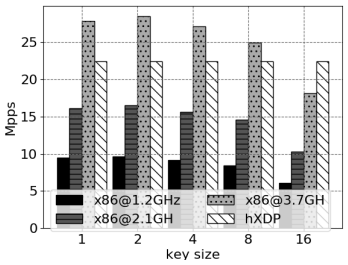

Figure 14: Impact on forwarding throughput of map accesses.

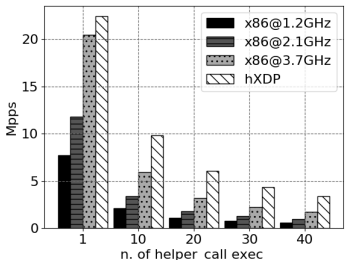

Figure 15: Forwarding tput when calling a helper function.

memory area and therefore did not incur in data consistency issues. Extending to more cores (lanes) would instead require the design of a more complex memory access system. Related to this, another interesting extension to our current design would be the support for larger DRAM or HBM memories, to store very large memory maps.

7 Related Work

NIC Programming AccelNet [18] is a match-action offloading engine used in large cloud datacenters to offload virtual switching and firewalling functions, implemented on top of the Catapult FPGA NIC [10]. FlexNIC [28] is a design based on the RMT [8] architecture, which provides a flexible network DMA interface used by operating systems and applications to offload stateless packet parsing and classification. P4->NetFPGA [1] and P4FPGA [51] provide high-level synthesis from the P4 [7] domain-specific language to an FPGA NIC platform. FlowBlaze [40] implements a finite-state machine abstraction using match-action tables on an FPGA NIC, to implement simple but high-performance network functions. Emu [45] uses high level synthesis to implement functions described in C# on the NetFPGA. Compared to these works, instead of match-action or higher-level abstractions, hXDP leverages abstractions defined by the Linux's kernel, and implements network functions described using the eBPF ISA.

The Netronome SmartNICs implement a limited form of eBPF/XDP offloading [29]. Unlike hXDP that implements a solution specifically targeted to XDP programs, the Netronome solution is added on top of their network processor as an afterthought, and therefore it is not specialized for the execution of XDP programs.

Application frameworks AccelTCP [34], Tonic [2] and Xtra [5] present abstractions, hardware architectures and prototypes to offload the transport protocol tasks to the NIC. We have not investigated the feasibility of using hXDP for a similar task, which is part of our future work. NICA [16] and ClickNP [32] are software/hardware frameworks that introduce specific software abstractions that connect FPGA blocks with an user program running on a general purpose CPU. In both cases, applications can only be designed composing the provided hardware blocks. hXDP provides instead an ISA that can be flexibly programmed, e.g., using higher level languages such as C.

Applications Examples of applications implemented on NICs include: DNS resolver [52]; the paxos protocol [46]; network slicing [53]; Key-value stores [30, 31, 44, 47]; Machine Learning [14, 20, 36]; and generic cloud services as proposed in [10, 41, 43]. [33] uses SmartNICs to provide a microservice-based platform to run different services. In this case, SoC-based NICs are used, e.g., based on Arm processing cores. Lynx [48] provides a system to implement network-facing services that need access to accelerators, such as GPUs, without involving the host system's CPU. Floem [38] is a framework to simplify the design of applications that leverage offloading to SoC-based SmartNICs. hXDP provides an XDP programming model that can be used to implement and extend these applications.

NIC Hardware Previous work did not focus on VLIW processors specialized for network processing on FPGA [25, 27]. [9] is the closest hXDP. It employs a non-specialized MIPS-based ISA and a VLIW architecture for packet processing. hXDP has an ISA design specifically targeted to network processing using the XDP abstractions. [19] presents an open-source 100-Gbps FPGA NIC design. hXDP can be integrated in such design to implement an open source FPGA NIC with XDP offloading support.

8 Conclusion

This paper presented the design and implementation of hXDP, a system to run Linux's XDP programs on FPGA NICs. hXDP can run unmodified XDP programs on FPGA matching the performance of a high-end x86 CPU core clocked at more than 2GHz. Designing and implementing hXDP required a significant research and engineering effort, which involved the design of a processor and its compiler, and while we believe that the performance results for a design running at 156MHz are already remarkable, we also identified several areas for future improvements. In fact, we consider hXDP a starting point and a tool to design future interfaces between operating systems/applications and network interface cards/accelerators. To foster work in this direction, we make our implementations available to the research community.

References

[1] P4-NetFPGA. https://github.com/NetFPGA/P4-NetFPGA-public/wiki.

[2] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff. Enabling programmable transport protocols in high-speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020.

[3] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, 1966.

[4] G. Bertin. Xdp in practice: integrating xdp into our ddos mitigation pipeline. In *Technical Conference on Linux Networking, Netdev*, volume 2, 2017.

[5] G. Bianchi, M. Welzl, A. Tulumello, F. Gringoli, G. Belocchi, M. Faltelli, and S. Pontarelli. XTRA: Towards portable transport layer functions. *IEEE Transactions on Network and Service Management*, 16(4):1507–1521, 2019.

[6] S. Biookaghazadeh, M. Zhao, and F. Ren. Are FPGAs suitable for edge computing? In {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[7] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, 44(3):87–95, 2014.

[8] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *ACM SIGCOMM '13*, ACM SIGCOMM '13, pages 99–110. ACM, 2013.

[9] M. S. Brunella, S. Pontarelli, M. Bonola, and G. Bianchi. V-PMP: A VLIW packet manipulator processor. In *2018 European Conference on Networks and Communications (EuCNC)*, pages 1–9. IEEE, 2018.

[10] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[11] V. Chamola, S. Patra, N. Kumar, and M. Guizani. Fpga for 5g: Re-configurable hardware for next generation communication. *IEEE Wireless Communications*, pages 1–8, 2020.

[12] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, Oct. 2018. USENIX Association.

[13] D. Chiou. The microsoft catapult project. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 124–124. IEEE, 2017.

[14] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, 2018.

[15] M. V. Dumitru, D. Dumitrescu, and C. Raiciu. Can we exploit buggy p4 programs? In *Proceedings of the Symposium on SDN Research*, SOSR '20, page 62–68, New York, NY, USA, 2020. Association for Computing Machinery.

[16] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein. Nica: An infrastructure for inline acceleration of network applications. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 345–362, 2019.

[17] Facebook. Facebook. 2018. Katran source code repository. https://github.com/facebookincubator/katran.

[18] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, 2018.

[19] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen. Corundum: An open-source 100-Gbps NIC. In *28th*

IEEE International Symposium on Field-Programmable Custom Computing Machines, 2020.

[20] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2018.

[21] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design principles for packet parsers. In *ACM/IEEE ANCS '13*.

[22] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied machine learning at Facebook: a datacenter infrastructure perspective. In *High Performance Computer Architecture (HPCA)*. IEEE, 2018.

[23] O. Hohlfeld, J. Krude, J. H. Reelfs, J. Rüth, and K. Wehrle. Demystifying the performance of XDP BPF. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 208–212. IEEE, 2019.

[24] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery.

[25] C. Iseli and E. Sanchez. Spyder: A reconfigurable vliw processor using FPGAs. In *[1993] Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pages 17–24. IEEE, 1993.

[26] S. Jiang, D. He, C. Yang, C. Xu, G. Luo, Y. Chen, Y. Liu, and J. Jiang. Accelerating mobile applications at the network edge with software-programmable fpgas. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 55–62. IEEE, 2018.

[27] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster. An FPGA-based VLIW processor with custom hardware execution. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 107–117, 2005.

[28] A. Kaufmann, S. Peter, T. Anderson, and A. Krishnamurthy. FlexNIC: rethinking network DMA. In *USENIX HotOS*, 2015.

[29] J. Kicinski and N. Viljoen. eBPF hardware offload to SmartNICs: cls bpf and XDP. *Proceedings of netdev*, 1, 2016.

[30] M. Lavasani, H. Angepat, and D. Chiou. An FPGA-based in-line accelerator for memcached. *IEEE Computer Architecture Letters*, 13(2):57–60, 2013.

[31] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. Kv-direct: High-performance in-memory key-value store with programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 137–152, 2017.

[32] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, and P. Cheng. Clicknp: Highly flexible and high-performance network processing with reconfigurable hardware. In *ACM SIGCOMM '16*.

[33] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana. E3: energy-efficient microservices on SmartNIC-accelerated servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, 2019.

[34] Y. Moon, S. Lee, M. A. Jamshed, and K. Park. Acceltcp: Accelerating network applications with stateful {TCP} offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 77–92, 2020.

[35] Netronome. AgilioTM CX 2x40GbE intelligent server adapter. https://www.netronome.com/media/redactor_files/PB_Agilio_CX_2x40GbE.pdf.

[36] K. Ovtcharov, O. Ruwase, J. Kim, J. Fowers, K. Strauss, and E. S. Chung. Toward accelerating deep learning at scale using specialized hardware in the datacenter. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–38, 2015.

[37] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung. Toward accelerating deep learning at scale using specialized hardware in the datacenter. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–38, 2015.

[38] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson. Floem: a programming system for NIC-accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, 2018.

[39] S. Pinneterre, S. Chiotakis, M. Paolino, and D. Raho. vfpgamanager: A virtualization framework for orchestrated fpga accelerator sharing in 5g cloud environments. In *2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pages 1–5. IEEE, 2018.

[40] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, et al. Flowblaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, 2019.

[41] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 13–24. IEEE Press, 2014.

[42] R. Ricart-Sanchez, P. Malagon, P. Salva-Garcia, E. C. Perez, Q. Wang, and J. M. A. Calero. Towards an fpga-accelerated programmable data path for edge-to-core communications in 5g networks. *Journal of Network and Computer Applications*, 124:80–93, 2018.

[43] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 150–156, 2017.

[44] G. Siracusano and R. Bifulco. Is it a SmartNIC or a key-value store? both! In *Proceedings of the SIGCOMM Posters and Demos*, pages 138–140. 2017.

[45] N. Sultana, S. Galea, D. Greaves, M. Wójcik, J. Shipton, R. Clegg, L. Mai, P. Bressana, R. Soulé, R. Mortier, et al. Emu: Rapid prototyping of networking services. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 459–471, 2017.

[46] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman. The case for in-network computing on demand. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[47] Y. Tokusashi, H. Matsutani, and N. Zilberman. Lake: the power of in-network computing. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2018.

[48] M. Tork, L. Maudlej, and M. Silberstein. Lynx: A smartnic-driven accelerator-centric architecture for network servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 117–131, New York, NY, USA, 2020. Association for Computing Machinery.

[49] W. Tu, J. Stringer, Y. Sun, and Y.-H. Wei. Bringing the power of ebpf to open vswitch. In *Linux Plumbers Conference*, 2018.

[50] F. J. Villanueva, M. J. Santofimia, D. Villa, J. Barba, and J. C. Lopez. Civitas: The smart city middleware, from sensors to big data. In *2013 Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 445–450. IEEE, 2013.

[51] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. P4FPGA: A rapid prototyping framework for P4. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 122–135, New York, NY, USA, 2017. ACM.

[52] J. Woodruff, M. Ramanujam, and N. Zilberman. P4DNS: In-network DNS. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–6. IEEE, 2019.

[53] Y. Yan, A. Beldachi, R. Nejabati, and D. Simeonidou. P4-enabled Smart NIC: Enabling sliceable and service-driven optical data centers. *Journal of Lightwave Technology*, 2020.

[54] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro '14*, 34(5):32–41, 2014.