eHDL: Turning eBPF/XDP Programs into Hardware Designs for the NIC

Alessandro Rivitti
alessandrorivitti@gmail.com
University of Roma Tor Vergata
Axbryd
Rome, Italy

Roberto Bifulco
roberto.bifulco@neclab.eu
NEC Laboratories Europe
Heidelberg, Germany

Angelo Tulumello
tulumello@axbryd.com
University of Roma Tor Vergata
Axbryd
Rome, Italy

Marco Bonola
mbonola@cnit.it
Consorzio Nazionale
Interuniverisitario per le
Telecomunicazioni
Rome, Italy

Salvatore Pontarelli
salvatore.pontarelli@uniroma1.it
Sapienza University of Rome, Italy
Rome, Italy

ABSTRACT

Scaling network packet processing performance to meet the increasing speed of network ports requires software programs to carefully leverage the network devices' hardware features. This is a complex task for network programmers, who need to learn and deal with the heterogeneity of device architectures, and re-think their software to leverage them. In this paper we make first steps to reverse this design process, enabling the automatic generation of tailored hardware designs starting from a network packet processing program. We introduce eHDL, a high-level synthesis tool that automatically generates hardware pipelines from unmodified Linux's eBPF/XDP programs. eHDL is designed to enable software developers to directly define and implement the hardware functions they need in the NIC. We prototype eHDL targeting a Xilinx Alveo U50 FPGA NIC, and evaluate it with a set of 5 eBPF/XDP programs. Our results show that the generated pipelines are efficient in terms of required hardware resources, using only 6.5%-13.3% of the FPGA, and always achieve the line rate forwarding throughput with about 1 microsecond of per-packet forwarding latency. Compared to other network-specific high-level synthesis tool, eHDL enables software programmers with no hardware expertise to describe stateful functions that operate on the entire packet data. Compared to alternative processor-based solutions that perform eBFP/XDP offloading to a NIC, eHDL provides 10-100x higher throughput.

CCS CONCEPTS

• **Networks → Programming interfaces**; • **Hardware → Hardware description languages and compilation**.

KEYWORDS

FPGA, HLS, eBPF, Network Programming, Hardware Offloading

ACM Reference Format:
Alessandro Rivitti, Roberto Bifulco, Angelo Tulumello, Marco Bonola, and Salvatore Pontarelli. 2023. eHDL: Turning eBPF/XDP Programs into Hardware Designs for the NIC. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3582016.3582035

1 INTRODUCTION

The per-port network speeds of server's network interface cards (NICs) is growing to over 100Gbps, requiring new system approaches to perform efficient network packet processing without overloading the host system's CPU. This is driving research in the per-application specialization of the entire networking subsystem, including the software network stacks [10] and NIC hardware [13, 16, 21, 28, 33, 35, 45].

The NIC data plane is a current focus for innovation, with many established CPU offload/acceleration features [24] being complemented by new *programmable* functions, such as programmable packet parsers and forwarding tables [3]. Nonetheless, most NIC designs are optimized for a selected set of large volume use cases, such as virtual switch offloading [36]. Other use cases are either not supported [39], or implementing them to leverage the new features requires a large ad-hoc engineering effort [7, 12, 29, 37, 38]. As a matter of fact, the recent large acquisitions of network hardware vendors in the semiconductor market are usually coupled with equally big, or bigger, investments on the software stacks required to leverage the new hardware at best [42].

What if software developers could directly define and implement the hardware functions they need in the NIC?

We notice that Linux recently included the ability to inject small programs into the kernel, using the eBPF technology, in order to customize the once fixed kernel's operations [22]. eBPF programs can be attached in different *hooks* in the Linux kernel and, among other uses, programmers use eBPF also to define packet processing tasks. eXpress Data Path (XDP) [19] is the hook in the earliest networking driver stage, i.e., before a packet is received by the kernel's network

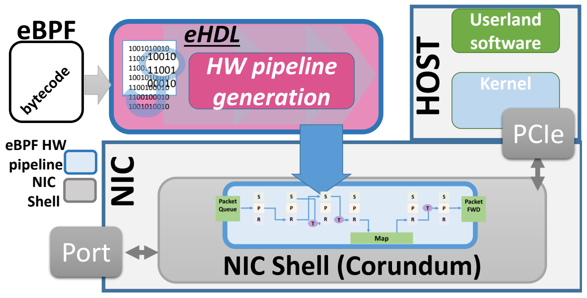


Figure 1: eHDL takes unmodified eBPF programs and generates tailored hardware to run them at line-rate

stack. For instance, network and service providers use XDP to implement load balancing [11], security [2], monitoring [18], deep packet inspection [41] and policy enforcement [8].

While purely in software, XDP programs effectively define processing tasks that are desired at the NIC level (although within its driver). We argue that such programs may in fact work as a formal specification of the needed hardware functions in the NIC, providing a straightforward interface for software developers to define their own NIC features.

Working towards such vision, we introduce eHDL, a high-level synthesis tool to turn eBPF/XDP programs into tailored hardware designs. eHDL solves three challenging issues: (i) turning a sequential program description into a (parallel) hardware implementation; (ii) transparently ensuring correct program operations, e.g., data consistency; (iii) and minimization of hardware resources requirements for the generated designs. To address these issues, we represent unmodified eBPF/XDP programs as a set of sequential transformations in a pipeline. Each pipeline's stage transforms the input packet and the program state, comprising eBPF's registers and stack, and provides the transformed data to the next stage.

eHDL uses the above representation to implement a hardware synthesis process in three steps: (i) program instructions parallelization; (ii) hardware primitives mapping; (iii) consistency handling and optimizations. During parallelization, we group the original eBPF program's instructions into schedules with varying degrees of parallelism, according to the program's control flow and data dependencies. Each group corresponds to a pipeline's stage, and each stage carries along a full replica of the current packet and program's state. We then define a library of hardware primitives to implement the eBPF instructions, and more complex eBPF data structures, such as *maps* (e.g., hash tables). During the second step, we combine such primitives to implement each stage's transformations. The resulting hardware design is a sequential pipeline: at any point in time there may be as many parallel program executions (and packets) as the number of stages. eHDL further processes this initial design during the third step, to include data consistency handling, and several optimizations that minimize the number of pipeline stages and hardware resources requirements. To optimize the pipeline we borrowed some well-know optimization techniques developed for microprocessor pipelines such as speculative execution or predication. The resulting hardware has only the features strictly required by the input program.

We prototype the hardware designs generated by eHDL using a Xilinx Alveo U50 100Gbps FPGA NIC (See Figure 1). FPGAs are a type of re-configurable hardware, and FPGA NICs are widely adopted in datacenter [6], 5G networks [32] and monitoring scenarios [31]. We test eHDL with five eBPF/XDP programs: the Linux's eBPF Router and Tunnel programs; an UDP firewall; the network security monitor Suricata [41]; and a dynamic DNAT. For all these applications, and when processing packets within the NIC, eHDL pipelines forward 100Gbps (line rate) with 64B packets, i.e., 148 Million packets per second (Mpps), with a per-packet forwarding latency of one microsecond. Furthermore, the generated pipelines use only 6.5%-13.3% of the FPGA hardware resources (Cf. Section 5). Compared to network-specific high-level synthesis tools, eHDL can describe stateful functions not supported by past works, and requires no hardware expertise, unlike more expressive, general purpose high-level synthesis tools. Compared to alternative processor-based solutions that perform eBFP/XDP offloading to a NIC (both in ASIC [33] and FPGA [5]), eHDL provides 10-100x higher throughput, depending on the implemented program.

In summary, our contributions are:

- a method to generate hardware pipelines starting from functions described by eBPF programs;
- eHDL, a High-level Synthesis tool that converts eBPF programs into RTL hardware descriptions;
- a detailed evaluation of the hardware pipelines generated by eHDL, including end-to-end tests with unmodified real-world eBPF programs and microbenchmarks.

2 BACKGROUND AND CONCEPT

Enabling software developers to define their own hardware functions requires to abstract away the hardware design process, making it *declarative*. This means that a developer should specify *what* a given task does, instead of defining *how* to do it. A convenient way to describe a task's goal is to provide a program that implements the task, but using a formalism that is familiar to the software developer, for instance asking the developer to write a C program. While the software program is in fact telling *how* to do the task, in the process it specifies unambiguously its goal. In this sense, past work has extensively explored the problem of taking a high-level program and turning it into a hardware design, in the field of High-Level Synthesis (HLS) [23, 30, 34, 43]. Given this large body of previous work, *why is* eHDL *needed at all*? The main reason is that all the previous works either limit the supported hardware functions to simpler (mostly) stateless packet header processing, or they require the programmer to have hardware design expertise.

The rest of this section presents related work and how eHDL extends them.

2.1 Related Work

Network HLS. The HLS tools dedicated to networking use cases usually adopt the P4 language [3], and target both FPGA [20, 43, 44] and programmable ASIC platforms [17, 25, 40]. P4 is a domain-specific language specialized to describe packet header parsing and classification tasks, and was developed in parallel with a reference hardware architecture called PISA, directly deriving assumptions and programming models from earlier work on programmable switching chips [4]. In fact, all the mentioned tools implement a PISA-like architecture for the hardware. A PISA-like architecture is especially suited for simpler packet header processing tasks (header parsing, classification) but has limited support for functions that need to keep state across the processing of multiple packets. This ultimately limits the functions a programmer can specify to mostly *stateless* packet classification tasks [29, 38]. For example, in Section 5 we could not implement a dynamic NAT function using the Xilinx SDNet P4 HLS compiler [44], due to the inability to specify a way to update the address translation tables required by the function.

General HLS. General purpose HLS tools [23, 30, 34], such as `Xilinx Vitis HLS` (Vitis) [46], allow programmers to describe hardware blocks in languages like C or C++, enabling the use of advanced language features like loops and classes. This makes hardware description faster than using traditional RTL languages (VHDL or Verilog). Furthermore, the tool can usually avoid a clock cycle-level hardware description, taking care of automatically defining the best allocation and scheduling strategy to meet area and throughput constraints [9]. Nonetheless, these tools assume that the programmer is in fact an expert in hardware design. That is, they do not abstract away the hardware design process, instead they provide a more convenient way to describe hardware. For example, Vitis uses `#pragma` statements to guide the compiler on how to handle the program (e.g., define datapath size, unrolling loops). Likewise, interfacing with the memories and hardware modules requires a deep knowledge on the possible choices between datatypes and IN/OUT protocols (e.g., Vitis makes extensive use of different flavours of the AXI4 protocol). This ultimately requires the programmer to specify in details the hardware behavior, including how to handle critical issues such as data consistency.

In short, we point out the main differences between eHDL and General HLS as follows:

- eHDL builds efficient pipelines that avoid timing closure problems without using hardware-related pragma/annotations.
- General HLS techniques usually work at data level, while eHDL works at a higher level of abstraction, i.e. at the packet level. This simplifies the design of network functions, and it is the level of abstraction commonly expected by network function programmers.
- eHDL automatically handles data consistency without (or with a very small) throughput degradation, implementing hazard avoidance techniques. Other HLS tools conservatively lower throughput regardless of the actual hazard, unless manual handling is performed.
- eHDL synthetizes hardware starting from bytecode (i.e., a sequence of RISC-like instructions), whereas existing HLS tools assume access to source code, e.g. C++. This makes the network programmer able to describe eBPF programs using different languages, e.g., C or Rust.

We present some examples in appendix A.4 to highlight the difference in user experience between eHDL and other HLS tools.

2.2 Why eBPF/XDP Helps?

Similarly to general HLS tools, programmers usually write eBPF programs using a high-level language, such as C. However, unlike general HLS tools, eBPF programs look like regular software,

```
1  int example(struct xdp_md *ctx) {
2  void *data_end = (void *)(long)ctx->data_end;
3  void *data = (void *)(long)ctx->data;
4  struct ethhdr *eth = data;
5  long *value;
6  int key = 0;

8  if ((data + sizeof(*eth)) > data_end)
9   return XDP_DROP;

11 if (eth->h_proto == ETH_P_IP)
12  key = 1;
13 else if (eth->h_proto == ETH_P_IPV6)
14  key = 2;
15 else if (eth->h_proto == ETH_P_ARP)
16  key = 3;

18 value = bpf_map_lookup_elem(&stats, &key);
19 if (value)
20  __sync_fetch_and_add(value, 1);

22 return XDP_TX;
23 }
```

Listing 1: A toy eBPF/XDP program example in C

although the eBPF/XDP environment enforces a specific programming model. This model is more constrained than general purpose languages, yet more flexible than network-specific languages such as P4. As we will see, this balance between constraints and flexibility is what enables eHDL to remove any hardware expertise requirements, while supporting unmodified eBPF/XDP programs that can easily express a large range of network applications. The eBPF limitations work on a twofold level: the computation performed in the kernel is both *time* and *memory* bounded. This corresponds to a set of limitations that are easily mapped to hardware constraints. **Time-bounded** means that the number of loops is given at compile time. In this way backward branches are only allowed in bounded loops so that they can be unrolled in a hardware pipeline. Similarly, **memory-bounded** means that there is no dynamic allocation, and the eBPF virtual machine memory is sharply split into two different types: (i) one memory is for per-packet operation, it is limited to the packet payload plus a limited temporary memory that the eBPF compiler manages as a stack; (ii) a memory that should persist across the processing of different packets. This memory is provided by eBPF using the concept of maps. To further clarify this aspect, we provide an eBPF/XDP program example in Listing 1.

The program counts received packets according to their Ethernet protocol's type field. We notice two important aspects: (i) the program receives as input an `xdp_md` struct, which contains pointers to the packet data buffer (Lines 2-3); (ii) the program accesses memory that is kept across executions (e.g., flow counters) using a *map* data structure (*stats*, defined outside the program scope) and *helper functions* (Lines 18, 20). Both aspects point to the programming model enforced by eBPF/XDP. A new program starts each time a packet is received, and the input and output of the

```
 0: r2 = *(u32 *)(r1 + 4)
 1: r1 = *(u32 *)(r1 + 0)
 2: r3 = 0
 3: *(u32 *)(r10 - 4) = r3
  [...]
 8: r2 = *(u8 *)(r1 + 12)
 9: r1 = *(u8 *)(r1 + 13)
10: r1 <<= 8
11: r1 |= r2
12: if r1 == 34525 goto +4
  [...]
19: r1 = 3
20: *(u32 *)(r10 - 4) = r1
21: r2 = r10
22: r2 += -4
23: r1 = 0 ll
25: call 1
26: r1 = r0
27: r0 = 3
28: if r1 == 0 goto +2
29: r2 = 1
30: lock *(u64 *)(r1 + 0) += r2
31: exit
```

Listing 2: eBPF bytecode for the program from Listing 1

program are the packet and related metadata (e.g., the transmit - *XDP_TX* - forwarding decision, Line 22). Any access to memory that persists across program executions uses helper functions and maps. This provides a way to clearly separate memory areas that live only within the program scope (e.g., key), from those that persist across different executions (e.g., the data pointed by value), and from those that should be moved towards next execution steps (e.g., the packet pointed by data). Furthermore, with the exception of the reading/writing to maps memory, the program has no other side-effects or dependencies. Therefore the evolution of each data variable, including the packet data, is fully specified within the program itself. In fact, helper functions might introduce side effects, but they are a fixed set of pre-specified functions with a fixed interface to exchange data with the eBPF program's context. This combination of factors allows us to make several assumptions about the control flow and data dependencies of the program, and therefore it makes possible to apply hardware design templates that can efficiently implement the program in a parallel fashion.

2.3 From eBPF to Hardware Pipelines

To explain how to derive hardware designs from XDP programs, we need a brief background about eBPF. eBPF programs run in the Linux kernel on the eBPF virtual machine: a RISC register-machine with 11 64b general purpose registers, 512B memory stack, and its own Instruction-Set Architecture (ISA). Registers and stack describe the entire current program state, whereas persisted state is located in *maps*. Programmers write eBPF programs using a high-level language, e.g. Listing 1, and then compile them to bytecode, i.e., a sequence of eBPF ISA instructions like the one shown in Listing 2.

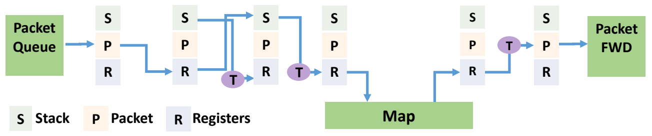


Figure 2: eHDL represents programs as a sequence of transformations on the packet and program state (register, stack). Maps are the only interface with external variables.

To turn eBPF bytecode into hardware designs, our idea is to describe the program in terms of state evolution, where the program's state is the combination of the packet data, and eBPF's registers and stack. In this abstraction each eBPF instruction becomes a transformation over the state so defined: an eBPF instruction reads from one of the memories areas and writes the result in a target memory area (See Figure 2). For instance, in Listing 2 instructions 8 and 9 move data among memories: they read a value from the packet memory and copy it in registers $R2$ and $R1$, respectively. Instruction 22, instead, reads a value from $R2$, transforms it performing a SUM with the constant -4, and writes the result back to $R2$.

Therefore, building on this intuition we can describe the program as a hardware design that effectively *unrolls* the program's instructions in a sequential pipeline, in which each stage contains a full copy of the packet data, registers and stack. The eBPF instructions define connections and operations between the i-th stage's memory areas and the $i+1$-th stage's memory areas. Since all the pipeline stages perform operations in parallel, this hardware design can potentially process in parallel as many packets as the number of pipeline's stages.

2.4 Challenges

The pipelined design described so far introduces two issues.

First, replicating the program state in each stage leads to inefficient use of hardware resources. For example, each stage requires over 2KB of memory: at least 1500B to hold a packet, 512B for the stack, and 88B for the 11 64b registers of the eBPF architecture. A single program can easily have in the order of hundreds of instructions and therefore as many stages. In real deployments, it is also possible that multiple XDP programs are loaded at the same time (e.g., to handle different types of protocols/traffic). This ultimately points to a need to reduce the per-stage memory resources and the overall number of stages.

Second, maps can introduce data consistency issues. In fact, maps memory can be read and written in different parts of the program. Since programs are unrolled as a pipeline in hardware, this effectively means that race conditions might arise. In particular, there are two possible data hazards: (i) a WAR (Write After Read) hazard, when the write operation erases the existing data too early; and (ii) a RAW (Read After Write) hazard in which the read operation reads stale data.

3 DESIGN

We design eHDL to work as a bytecode-to-source compiler. It takes as input unmodified eBPF bytecode and outputs HDL (VHDL). The generated HDL is e.g., ready for integration in an FPGA NIC *shell*,

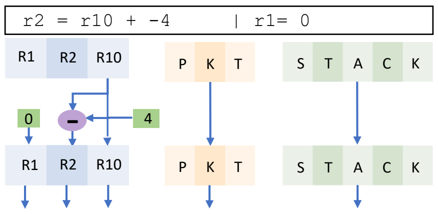


Figure 3: Register to register primitive

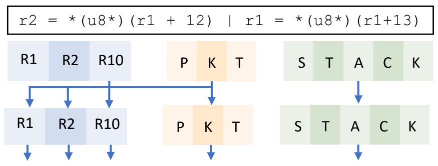


Figure 4: Load packet to register primitive

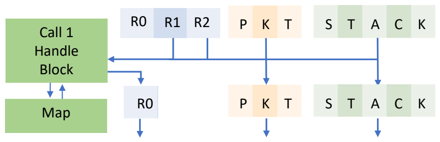


Figure 5: Helper functions blocks: $R1$-$R5$ are input parameters, $R0$ is the return value. Optional read/write to the packet data is supported when needed

which provides access to I/O interfaces and PCIe bus, using the target FPGA compiler toolchain.

To address the challenges outlined in Section 2, eHDL performs compile-time analysis and optimization. First, eHDL statically analyzes the input bytecode to extract any instructions dependencies. This enables to extract instruction-level parallelism (ILP) and to track what subset of the program state is actually required at any point in the execution of the program. In the process, eHDL tracks the memory areas accessed by the instructions, i.e., Stack, Packet or Map, and labels each instruction accordingly. Second, eHDL applies instructions fusion, e.g., using three-operand instructions in place of two-operand ones. The outcomes of these first steps are: (i) the program's Control-Flow Graph (CFG); (ii) the Data-Dependency Graph (DDG); and (iii) a parallel schedule of execution of the labeled program's instructions. Each row in the schedule contains instructions that can be executed in parallel and corresponds to a pipeline's stage. With the logical program's pipeline laid out, eHDL then proceeds to: (i) map logical instructions to hardware "template" blocks; (ii) enforce the program control flow on the pipeline; (ii) handle data consistency issues for map memory (§ 4.1); and (iv) minimize the size of the program state applying packet framing (§ 4.2) and register pruning (§ 4.3).

The result of these steps is a hardware design that supports the full eBPF ISA, helper functions and maps, while addressing the resource minimization and data consistency issues outlined earlier. The remainder of this section presents the steps to build the complete pipeline, while we defer to Section 4 the discussion about maps support and optimization.

3.1 Program Analysis and Instruction Labeling

As a first step, eHDL builds the program's CFG and DDG. Then, traversing these two graphs, it inspects every program's load and store instructions to identify the type of memory area read/written by each of them. In fact, eBPF includes three types of memory: the program's stack; the packet buffer; map memory. Furthermore, each map defined within the program has its own dedicated address space. eHDL assigns a label corresponding to the related memory area to each instruction. The analysis is performed as follows.

First, eHDL tracks the use of $R10$, which is a read-only eBPF register that contains the stack pointer of the program. The load/store instructions reading $R10$ for the target memory address are accessing the stack. Using the CFG and DDG, eHDL then tracks all the downstream variables that contain values derived from $R10$ (e.g., `r9 = r10 +10`). All the instructions using these values are labeled as *stack memory*. Second, eHDL tracks the use of $R1$, which is set at program start to point to the `xdp_md` struct. The struct contains the packet buffer address (`xdp_md->data`) that is usually read at the beginning of the program. Like in the previous case, eHDL performs register dependency analysis to label instructions that use the packet buffer address (and derived variables) as *packet memory*. Finally, eHDL tracks the use of $R0$, which holds a pointer to a specific map after every call to `bpf_map_lookup_elem()`. In this case, the register dependency analysis is used to label instructions with *map memory* labels that are specific to each map.

As we will see next, the memory type labels play an important role during the generation of the hardware pipeline, since they identify hardware primitives that should be instantiated. Likewise, they also allow eHDL to identify data hazards.

3.2 Instruction Fusion and Transformation

The direct implementation of eBPF instructions in the pipeline allows us to easily extend the eBPF ISA to fuse multiple instructions in one. For instance, we could use three-operand operations in place of two-operand ones [5]. That is, we can *generate* a given instruction when needed.[1]

More generally, we remark that in a processor implementation expanding the ISA requires to study the trade-off between the extended instruction's resource requirements and the effective gain (e.g., how often the new instruction is used). For example, in hXDP [5], a processor tailored for eBPF/XDP programs execution, the use of 6-bytes load and store instructions is useful only when the eBPF network function modifies the MAC addresses, otherwise it creates just resources overhead. Instead in eHDL there's no trade-off to be evaluated when fusing instructions since we deploy the associated hardware architecture for the new ISA instructions only when needed.

3.3 Parallelization

The parallelization step leverages instruction-level parallelism to execute multiple instructions in the same clock cycle. This has the potential to reduce the number of stages in the pipeline, since each stage could execute more than a single instruction. We analyze instruction-level parallelism using the CFG and DDG: two instructions can be executed in parallel if they belong to the same control block, and if they have no data dependencies among each other.

Like when defining new ISA instructions, the ad-hoc generation of hardware designs of eHDL allows us to avoid trade-off decisions

[1]The new instructions should not limit the pipeline clock frequency. Aggressive instructions fusion might require additional mechanisms to balance the instructions-fusion and the maximum pipeline clock frequency.

that are typical in fixed designs. In the case of parallel executions of multiple instructions, it is common to trade-off between the achievable parallelism and the hardware resource requirements. In particular, providing hardware to execute a small number of parallel instructions constrains the level of parallelism. In contrast, more hardware to run instructions in parallel will lead to often idle (i.e., wasted) subsystems, since control and data dependencies cause programs to have varying degrees of instruction-level parallelism during their execution. The hardware pipelines generated by `eHDL` are tailored to a specific program, so the degree of parallelism can grow and shrink in each pipeline's stage. That is, when a set of instructions can run in parallel, `eHDL` expands the stage to run all of them in that stage. Conversely, if a next instruction cannot be executed in parallel with any other instruction, `eHDL` will generate a pipeline stage that has only the amount of resources required to run a single instruction.

3.4 Template Hardware Primitives

The parallelization step provides a logical pipeline, which has to be translated into a corresponding hardware pipeline. We perform this step mapping each instruction to a set of hardware primitives that implement the individual transformations. Depending on the type of instruction (register to register, load/store, call/jmp) and the instruction label (stack memory, packet memory, map memory), we instantiate one or more different hardware primitives between two pipeline's stages. Next we detail the main primitives.

3.4.1 Simple instructions. **Register to register.** Instructions that use immediate values and/or the $R0$-$R10$ registers (e.g., `add`, `sub`, `and`, etc.) take source registers from the i-th stage and the destination register is the corresponding element of the array of eBPF registers of the $(i+1)$-th stage (See Figure 3 for an example).
Load and store. Instructions that load/store from/to the stack or packet memory can be implemented as a direct connection between elements of the respective register arrays. Figure 4 shows the mapping of two load instructions. These instructions are implemented by simply connecting an element of the packet array at the i-th stage with the $R1$ and $R2$ registers of the $(i+1)$-th stage. Store instructions work in a similar way, connecting the i-th stage of the register array with the $(i+1)$-th stage of the packet or stack array.

3.4.2 Helper Functions. Helper functions are outside of the bytecode scope, and they are defined within each kernel version. This allows us to efficiently implement each relevant helper function as a custom hardware block.[2] Figure 5 shows an example of the insertion of a helper function as a dedicated hardware block between two pipeline's stages. In particular, we defined a common hardware interface for all helper functions blocks, mimicking the eBPF helper function's software interface: (i) $R1$-$R5$ provide the input parameters, therefore they feed the corresponding function's hardware block; (ii) the function's output is connected to the next stage's $R0$; (iii) two optional inputs, depending on the specific helper function called, that can read both the stack and packet frames; and (iv) an optional output that will write the packet frames if the helper function writes to the packet buffer. The helper function block can be implemented itself in a pipelined manner, with a variable number of stages between input and output depending on the complexity of the function. We highlight that multiple calls to the same helper function will generate multiple instances of the same hardware block, replicated in different pipeline stages in order to avoid contention on the hardware resources.

[2]Several helper functions such as `bpf_get_smp_processor_id()` are meaningful only for a CPU implementation, we provide a stub for those.

3.5 Control Flow Enforcement

Finally, to enforce the program control flow we exploit an approach similar to predication [27] used in microprocessors. `eHDL` generates a set of control signals to enable/disable pipeline's stages according to the result of `goto/jump` instructions. That is, each packet traverses all the pipeline stages independently from the specific control flow path for that packet, however, only enabled stages actually perform operations, whereas the disabled ones just move the packet to the next stage.

The parallelization step described in Section 3.3 ensures that all backward jumps are replaced with forward jumps, in order to ensure that the entire program can be described as a strictly forward-feeding pipeline. This is always possible, since eBPF forbids unbounded loops.

4 CONSISTENCY AND OPTIMIZATIONS

A design generated by `eHDL` as described in Section 3 still does not include access to maps, and requires significant hardware resources, since it replicates packet and program state in each stage. This section describes how we address maps access and data consistency, and the packet framing and state pruning techniques we apply to optimize the pipeline.

We conclude the section providing a complete example of the hardware design generated for our running example from Listing 1, and how it can be readily used with FPGA NICs.

4.1 eBPF Maps and Data Consistency

Maps are statically created when the eBPF program is first loaded. `eHDL` identifies the maps' parameters specified at compile time (such as the number of entries and key size) and instantiates a suitable hardware block called `eHDLmap` that implements the interface towards the map memory. Here, notice that maps are accessed both with the helper functions and with load/store instructions targeting map memory.

Two specific helper functions perform *lookups* and *updates* to the maps' entries. These differ from helpers described in §3.4.2, since their interface with the pipeline's stages includes: $R1$ and a portion of the stack, which is used as input (to store the `key` value for the lookup/update); and $R0$, which is used as output. Another key difference is that multiple calls of these helper functions to access the same map correspond to multiple accesses to the same hardware block. That is, while with other helper functions the hardware block is always replicated for multiple function calls, helper functions accessing the same map share a common hardware block across pipeline's stages. Thus, all the instances of the `eHDLmap` block for a given map are in fact interfaced with a single memory area. Load and Store instructions that access a map memory are also connected to the corresponding `eHDLmap` block. Our current hardware primitive supports multiple read-write channels to enable parallel

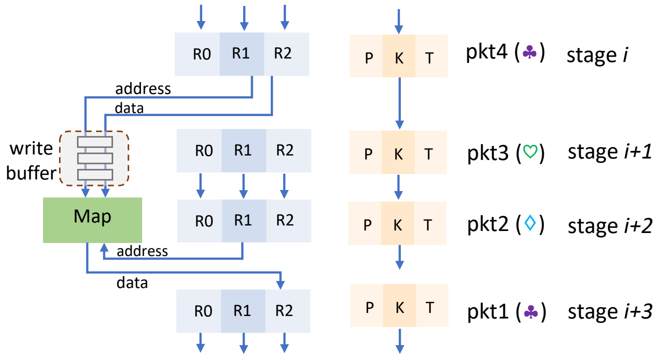


Figure 6: WAR: pkt4 writes in the location read by pkt1

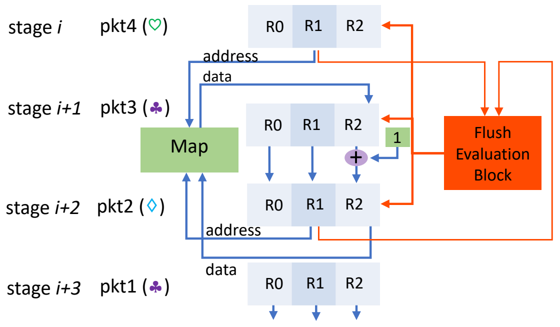


Figure 7: RAW: pkt4 reads the same location in which pkt2 writes

access to the memory from different pipeline stages. Nonetheless, in all the examined use cases at most two memory channels to the same map were needed. It is important to notice that in all cases the memory area also supports a suitable interface towards the host machine, to enable communication with the host using the standard eBPF tools, e.g., to read/write to the map from userspace.

Data consistency. As discussed already in Section 2, WAR and RAW data hazard might occur whenever multiple pipeline stages access the same map memory. However, the probability of facing actual data consistency issues vary significantly. In particular, we observe that there are two main uses of eBPF maps, in line with classical approaches in designing network functions [38]. In a first case, the eBPF map stores data related to a network flow, i.e., *flow state*. That is, the key that identifies the corresponding map entry in this case is a flow identifier. This case creates data hazards, but only when packets belonging to the same flow are processed in short sequence within the pipeline. In a second case, the eBPF maps are accessed with keys that are not related to a network flow identifier, i.e., *global state*. For instance, this is the case when an eBPF program keeps counters, like in the example from Listing 1. In this case, the data hazards may happen with high probability, since the pipeline often accesses the same few entries, and potentially for each processed packet.

4.1.1 *Handling WAR.*

WAR hazards are handled in both cases with the same approach. We describe our solution using the example shown in Figure 6. There are four packets traversing in the pipeline, and the packets belong to three different flows (♣,♡ and ♢) each accessing a different memory address. Since *pkt2* and *pkt4* belong to the same flow (♣), they access the same memory address. This would cause a WAR hazard: the $(i + 3)$-th stage will read the value just written by the processing happening at the earlier (i)-th stage. To prevent this issue, eHDL adds some additional registers (the shaded box in Figure 6) that delay the actual writing of the value computed in the i-th pipeline stage, until the reading of the $(i + 3)$-th stage is finalized.

4.1.2 *Handling RAW.*

Handling RAW hazards can significantly benefit from the distinction between the flow and global state cases. In fact, in the first case we flush the pipeline to maintain consistency. This approach is similar to speculative execution, in which the misspeculation corresponds to the case in which multiple packets belong to the same flow. In the second case we utilize specific atomic operators.

Per-flow state. In Figure 7 we show a RAW hazard in the case of a per-flow eBPF map entry. In this case, the problem is due to the reading of the value from the reading stage (the (i)-th stage), before writing the updated value from the writing stage (the $(i + 2)$-th). Whenever this hazard is detected, we flush the pipeline discarding any intermediate computation. The detection of this hazard is done by a dedicated hardware block, the `Flush Evaluation Block` (red block in Figure 7). The block stores the sequence of memory addresses of unconfirmed read operations and, if one of these addresses is written, it issues a pipeline flush signal. In the example, the `Flush Evaluation Block` stores the addresses corresponding to *pkt3* and *pkt4* (say the ♡ and ♣ addresses) and compares these values with the address to be written by *pkt2* (♣) in the writing stage. Since the addresses for *pkt2* and *pkt4* are the same, a hazard is detected.

Flushing the pipeline requires to repeat the execution of all the packets from the beginning of the pipeline up to the read stage, which reduces the pipeline throughput. Thus, we evaluated alternative approaches, such as the one adopted by FlowBlaze [38], based on pipeline stalling. In place of flushing, stalling introduces pipeline "bubbles" while waiting for the memory write. Unfortunately, this requires to identify the hazard during the reading stage, which is only possible if the writing address can be inferred in advance. We therefore decided to keep our approach with flushing as a single generic solution. Nonetheless, as we will show in Section 5, this does not significantly affect the throughput performance in real cases. In fact, in practical cases the probability of a flush is relatively low, which allows the pipeline to *amortize* the clock cycles spent to repeat flushed computations.

Global state. When programs use maps to access a value with high frequency, flushing the pipeline severely degrades throughput. This type of access to maps introduces performance issues also in the software implementation of eBPF, in which case the update requires synchronizing memory accesses across the multiple CPU caches. Nonetheless, in many use cases this type of access pattern is typical to update variables such as counters, e.g., to collect aggregated traffic statistics. For such cases, to avoid expensive locking of the memory areas eBPF introduces atomic operations.[3] An example

[3]https://reviews.llvm.org/D72184

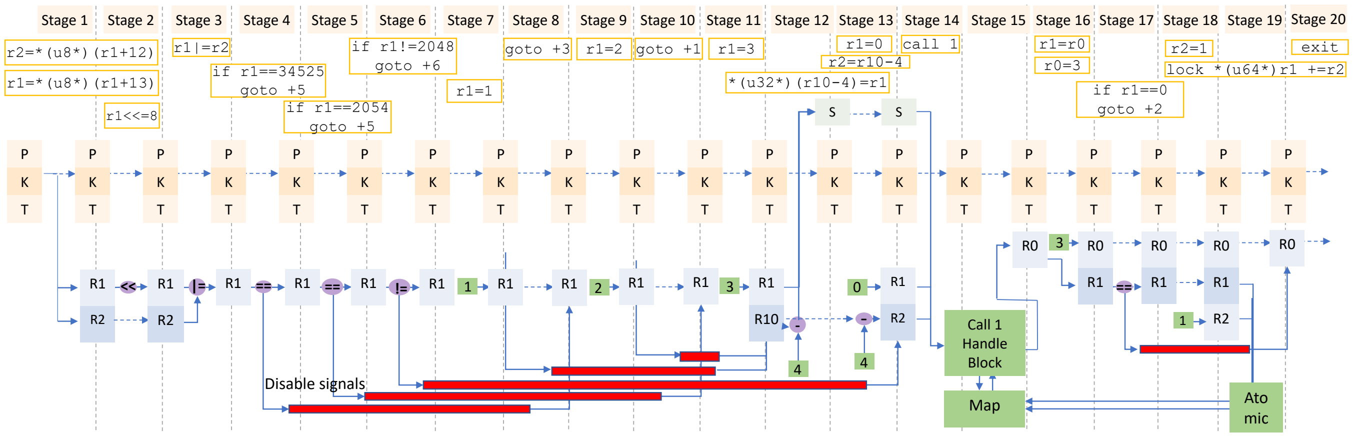


Figure 8: Design generated by eHDL for the input program bytecode reported in Listing 2

of such operations is in Listing 1, Line 22. Since programmers already use this construct to improve the performance of their eBPF programs, we can readily re-use these as is, and introduce specialized hardware primitives that perform atomic operations on map memory in an efficient manner. That is, this block performs key-based lookup and value update *in place*, without copying back the data to the pipeline's register and stack memories. Conceptually, these atomic operation primitives are similar to the ones proposed in [40], for atomic operations on the registers of a switch data plane.

4.1.3 Multiple Hazard Occurences. Whenever multiple reads to a single MAP are requested by a program, the buffer deployed to overcome the WAR hazard is long enough to enable the last pipeline stage that requests a read to actually perform a read on the previous value. This is an easy solution with a relatively low impact on area utilization. However, for what concerns RAW hazards, in order to enable multiple writes to be performed after a MAP read we need to instantiate a Flush Evaluation Block for every single map write instruction. In this way, we can evaluate the hazards separately and whenever a need for flushing is requested by either block, the pipe will be flushed up to the requested pipeline stage. Of course, the area utilization for this approach is not fully optimized however we found that in the use cases evaluated, this need for multiple Flush Evaluation Blocks for a single map read was never needed.

4.2 Packet Framing

In the generated pipeline as described so far, we assume that the entire packet buffer is propagated in each stage. In practice, doing so would be wasteful. In fact, a large share of network packets is much smaller than the maximum packet size observed in the network. For instance, in networks with a maximum packet size of about 1500B, it is common to observe much smaller average packet sizes at 700B or less. This is a common problem in any network hardware design, and the typical solution is to chunk the packet in frames of smaller size [48]. For example, typical frame sizes are 32B [47], and 64B [15]. In the designs generated by eHDL, framing the packet introduces an additional factor of complexity, since an eBPF program can theoretically access any part of the packet at any time. That is, the processing performed by one of the pipeline's stages might need to access a packet frame that is located in a different pipeline stage. To address the issue, eHDL enables connections between a stage's register and the packet frame located in previous stages. Here, it should be noted that the frames located in previous stages are simply *propagated* through the pipeline, since such stages are in fact *disabled*. That is, processing in the pipeline advances synchronously with the first packet's frame. Also in this case, the used approach is similar to a technique used in microprocessor pipelines, *i.e.* stage bypassing: the data coming from previous stages are anticipated to the stage of the first frame of a packet to mantain data consistency.

A special case is when the frame to be accessed is not yet in the pipeline by the time the processing requires it. For example, this may happen if an instruction at the beginning of the program accesses data at the end of a large packet. eHDL handles these cases by introducing synthetic NOP stages, with the only goal of making the pipeline longer.

It is worth noting that actual network functions rarely go deep into the payload, so the hardware complexity is easy to manage. During the development of eHDL, we discussed the possibility of deploying elastic buffers to face the problem of accessing the entirety of the packet within each pipeline stage, however we did not develop such a solution since we didn't see concrete use cases in which we need to use this block.

4.3 State Pruning

Program state, i.e., registers and stack, is also replicated in each stage. However, at any point the program uses only a subset of the registers and stack. This is connected to the lifetime of the defined variables. For example, in Listing 2, $r2$ is used last time in instruction 11, and then re-assigned in instruction 21. Therefore, between instructions 11 and 22, there is no need to keep the value of $r2$. eHDL performs a similar analysis for all the registers and the stack addresses. The resulting pipeline will have only the required registers and stack memory in each of the stages.

4.4 Pipeline Example

In Figure 8 we show the high-level hardware design generated by eHDL for the program from Listing 1. A few things can be pointed

Table 1: Applications used for evaluation

Program	Description
Simple firewall	checks the bidirectional connectivity for UDP flows
Tunnel	parse pkt up to L4, encapsulate and XDP_TX
Router	parse pkt headers up to IP, look up in routing table and forward (redirect)
DNAT	an application performing dynamic source NAT
Suricata	an Intrusion Detection System (IDS) [41]

out. First, instructions corresponding to program Lines 8-9 are not present, since they are used to check that memory accesses are always performed within the packet data length. This check is readily implemented in hardware when accessing the packet frame, and it can be therefore safely skipped. Second, the instruction-level parallelism is at most two, and only few instructions can be actually parallelized (See stages 1, 11, and 16). This is because our running example is heavy on control flow, as demonstrated by the several disable signals (the red bars in Figure) corresponding to the `if` statements in the program. Finally, we can observe that state pruning effectively reduces the number of registers and stack memory propagated throughout the pipeline. Most of the stages (9) only have a single 8B register, 6 stages have 2 registers and only one stage has 3 registers. Without pruning, each stage would have 11 registers. Similarly, stack memory is only present in 2 stages out of 20, and it is only big enough to hold the key required for the lookup in the array map (4B in place of 512B). All combined, the largest of the stages only requires 88B of memory (assuming 64B packet frame + 24B for 3 registers) in place of the over 2KB it would have required without packet framing and state pruning.

4.5 Integration in the NIC *Shell*

eHDL integrates automatically the generated pipeline in a target NIC *shell*, which connects to the system I/O (e.g., network ports and to the host's PCIe bus). Without loss of generality, we target the Corundum 100 Gbps NIC [15] for our prototype. We remark that eHDL wraps the generated designs into a set of asynchronous FIFO queues, in order to decouple them from the *shell*, giving the option to use different clock frequencies for the *shell* (e.g., clocked at 250 MHz) and the pipeline.

5 EVALUATION

We evaluate eHDL measuring the performance (throughput, latency, energy) of the generated hardware designs, and their resource requirements, and performing a set of microbenchmarks to assess the impact of the design decisions discussed throughout the paper. Finally, we qualitatively compare the eHDL programming experience with the one provided by HLS tools, when implementing the program from Listing 1.

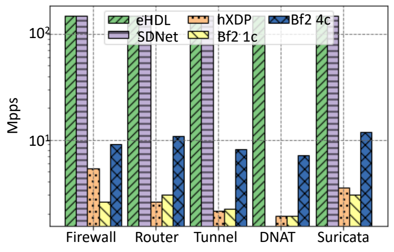
Testbed. eHDL designs are implemented targeting a 100Gbps Xilinx ALVEO U50 NIC. To test the system end-to-end we use two directly connected machines. One is equipped with a 100Gbps Mellanox ConnectX-5 NIC, and runs a DPDK traffic generator capable of generating line rate traffic with 64B packets (i.e., 148 Mpps). The other machine hosts the Xilinx Alveo U50, or an NVIDIA Bluefield2 DPU, depending on the test. In all tests we measure the performance at the traffic generator system, counting received packets for throughput tests and using hardware timestamping for latency tests. Depending on the test, we vary the number of generated flows from 1 to over 100k. For some microbenchmarks we instrument ad-hoc hardware counters into the FPGA. All our tests measure the performance of applications running entirely within the NIC, and do not transfer packets to the host system. In fact, in that case the PCIe bus transfers and the operating system NIC's driver would become the bottlenecks.

Comparison terms. We compare hardware designs generated by eHDL against those generated by the **Xilinx SDNet P4** High-level Synthesis compiler [44], which synthesizes hardware pipelines from P4 programs [3]. The hardware pipelines target the same Xilinx Alveo U50 used for eHDL. We also compare against running eBPF programs on **hXDP** [5] and on an **NVIDIA Bluefield2**. To the best of our knowledge, these are the currently available solutions that support running full eBPF programs on the NIC, and they are both processor-based. We use the latest version of hXDP, synthetized on the same Alveo U50 used for eHDL. hXDP implements a single core, 2 lanes Very-Long-Instruction-Word processor, clocked at 250MHz. The Bluefield2 (Bf2) combines two main subsystems: a switching data plane based on the Mellanox ConnectX6 architecture; and a battery of 8 general purpose Arm A72 cores running at up to 2.75GHz. The ConnectX6 receives the packets from the network ports and can forward them directly to the host system, like a regular NIC, or it can re-direct them to the Arm CPUs. In the latter configuration, the Bf2 can run eBPF programs directly on the NIC, leveraging the Arm CPU.

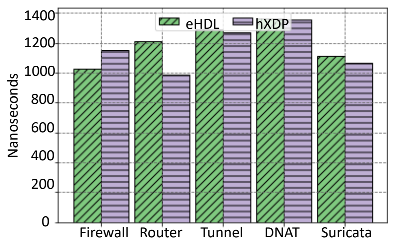
Test Applications. We use 5 unmodified real-world eBPF applications as input to eHDL. We summarize them in Table 1. The **Tunnel** and **Router** applications are the Linux's XDP applications included with the kernel sources (`tx_iptunnel` and `router_ipv4`, respectively). Both applications use *global state* to keep aggregated traffic statistics. **Simple Firewall** is an application that tracks the bi-directional connection establishment for flows defined by the 5-tuple. **DNAT** is an implementation of a dynamic source Network-Address-Translation application. On the first packet of a flow, the DNAT selects a new port/address combination. Any following packet belonging to the bi-directional flow gets the source/destination address/port accordingly translated. The port selection is performed directly in the data plane, requiring read/write access to the eBPF maps. **Suricata** is an open source Network Intrusion Detection System, which generates eBPF programs to filter traffic as early as possible [41]. In addition to parsing packets and checking access control lists, Suricata keeps also track of aggregated traffic statistics using *global state*. To compare against SDNet, we port the eBPF programs for Simple Firewall, Router, Tunnel and Suricata to equivalent P4 implementations. We could not implement the DNAT in P4 [3], since there is no obvious way to define the dynamic port selection within the data plane with SDNet P4.

5.1 Throughput and Latency

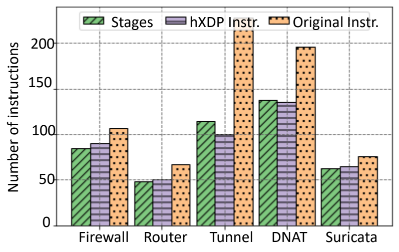
Figure 9.a (notice the log scale) shows the throughput measured in our tests when generating 10k flows and 148Mpps (100Gbps). All the hardware pipelines generated by eHDL can forward 148Mpps. SDNet can also forward 148Mpps, however it cannot implement the DNAT. hXDP can only forward 0.9-5.4Mpps depending on the



(a) Throughput

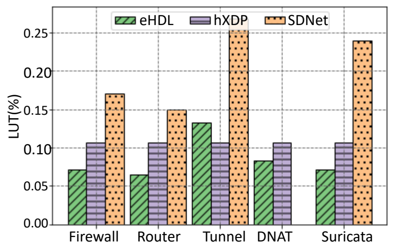


(b) Latency

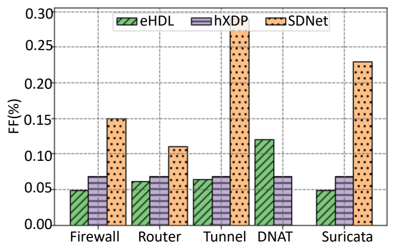


(c) Instructions vs Number of Stages

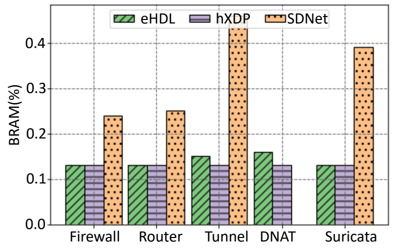
Figure 9: Performance results and comparison between number of eHDL pipelines' stages and eBPF instructions. Bf2 is an NVIDIA Bluefield2 using 1 (1c) or 4 (4c) of its Arm cores. eHDL provides 10-100x higher throughput than hXDP and Bf2, and it is more flexible than SDNet P4, which for instance cannot implement DNAT



(a) LUTs



(b) Flip-Flop



(c) BRAM

Figure 10: FPGA resource requirements (on Xilinx Alveo U50). eHDL designs are comparable or more efficient than hXDP, and significantly more efficient (2-4 times lower requirements) than SDNet. This result is due to eHDL's ability to tailor the design to the use case. The hXDP resources are the same for all use cases, since it is a processor-based design

use case. The Bf2 is comparable to hXDP when using a single Arm cores (Bf2 1c), or slightly faster, growing linearly to over 10Mpps when using multiple cores (Bf2 4c shows the result for 4 cores).

Figure 9.b shows the forwarding latency only for eHDL pipelines and hXDP, for readability, since their latency results are 10x lower than those of the Bf2, and thus directly comparable among each other. In both cases, the latency is remarkably low for all use cases, at about 1 microsecond.

The reported latency variation is explained looking at Figure 9.c, where we report the number of stages for the eHDL pipelines, vs the number of instructions of hXDP, and of the original bytecode. The use cases with less pipeline stages experience lower latency. We can also observe that both eHDL and hXDP can reduce the number of original instructions, sometimes by about 50%. The latency of eHDL and hXDP is in fact comparable since they both leverage instruction-level parallelism in the same way. The small differences between number of stages and number of hXDP instructions are due to some architecture specific variations (e.g., eHDL might add stages to implement helper functions in-line). However, the throughput of eHDL pipelines is much higher since packets are processed in parallel within the pipeline, whereas packets in hXDP are processed one by one. We want to remark that we limited the testbed to a 250MHz frequency to match the 100Gbps scenario. We believe that with extensive optimization higher frequencies are easily achievable and thus higher rates too. Anyway, when optimizing to increase the frequency there is the risk of reducing usability. In fact, excessive optimization might lead to cases in which timing closure is not always possible for the generated hardware.

5.2 Hardware Resources and Energy

Figure 10 shows the FPGA resources occupation for logic (LUTs) (a), Flip-Flops (b) and BRAM (c). All the results include the Corundum resources. eHDL requirements are roughly comparable to hXDP, and significantly lower than those of SDNet-generated designs. This is not surprising since SDNet instantiates generic programmable parser and lookup tables [26]. eHDL generates instead designs that are strictly tailored to the input program, hence very efficient.

We then measure the overall power absorption of the machine under test during throughput tests, and for one minute of test, while the machine's CPU was idle in lowest power state. Pairing this with the throughput results from Figure 9 helps in getting a rough estimate of energy requirements, even if we have no visibility in all NIC subsystems. We measured on average 80-85W when the system under test hosts the Xilinx Alveo U50, with little variation when the FPGA is flashed with eHDL, hXDP or SDNet hardware designs. The same machine consumes 100-105W when hosting the Bf2.

5.3 Impact of Flushing

eHDL pipelines are clocked at 250MHz and can forward a packet in each clock-cycle, achieving a theoretical 250Mpps throughput. This makes it hard to see the impact of flushing in the end-to-end tests

Table 2: Packet loss and number of flush events for a leaky bucket application with different real-world traces

Trace	# lost packets	# flushes
Caida	0	350k/sec
Mawi	0	124k/sec

presented so far. We therefore developed an application that has a high chance of performing flushes, and instrumented the generated pipeline with counters for flush events. The application is a Leaky Bucket, which needs to track the time of reception of each packet to check the packet forwarding rate. This leads to RAW hazards that cannot be solved with atomic operations and thus to flush events. We then used traffic traces to test the system under realistic flow distributions. We use two traces: one from CAIDA [1] (caida_20190117-134900); and one from MAWI [14] (mawi_202103221400). The traces have an average packet size of 411B and 573B, and 184'305 and 163'697 5-tuple flows, respectively. Table 2 reports the results in terms of number of lost packets and number of flushes per second, when replying the traces at 100Gbps. Also in this case and as expected, the eHDL pipeline does not drop any packets. The combined effect of larger than minimum packet sizes and a realistic distribution of flows is such that the probability of flushing is low. In fact, we measure a relatively low number of pipeline flush events per second (in any case below 350k). For reference, when replicating the CAIDA trace, but changing the eBPF program to access always the same map address (i.e., like if all the packets were part of a single flow), the maximum achieved throughput degraded from the 29Mpps to 12Mpps (i.e., about 40Gbps), showing the effectiveness of our data hazard handling method. For further analysis, we present a detailed analytical study of flushing in Appendix.

5.4 Impact of State Pruning

We show the impact of the state pruning optimizations, using the pipeline from Figure 8 and comparing it to a pipeline generated for the same program, but with state pruning disabled. In these results, we count only the hardware resources required by the pipeline (i.e., without the overhead of Corundum). The pipeline without state pruning requires 46%, 66% and 123% more LUTs, Flip-Flops and BRAM than the pipeline with pruning enabled, respectively.

5.5 Programming Experience

Finally, we describe the differences between the eHDL and Vitis HLS workflows, to clarify why current HLS tools are not a suitable solution for network function developers.

Coding the program in Listing 1 to make it work with Vitis HLS required a hardware expert to define I/O interfaces and pragmas. We report the complete source code in Appendix, while here we point out to some examples.

The function inputs are of type stream<axiWord>, i.e., the programmer is required to implement state machines to reconstruct the packet data that is chunked in frames, and be knowledgeable with the AXI stream protocol, e.g., handling control signals such as valid and ready. On the contrary, with eHDL it is possible to work directly at packet level. Variables need to be annotated with several pragmas, such as:

```
#pragma interface mode=axis port=ethStreamIn
#pragma HLS STREAM variable=key depth=256
```

These tell how to connect to other modules within the hardware design, and how to drive specific hardware design choices, such as type of memory resource and datapath width. The implications of these choices require deep hardware expertise. In contrast, with eHDL the code from Listing 1 is all that is needed.

In terms of toolchain, eHDL starts from the eBPF bytecode generated by the compilation of the program in Listing 1, and generates the firmware ready to be loaded on the Xilinx U50. In contrast, with Vitis HLS the programmer has to: (i) (re)implement the network function in C++ using the proper hardware annotations; (ii) generate an IPcore using HLS; (iii) manually connect the IPcore to the NIC shell to implement the end-to-end FPGA hardware design.

6 DISCUSSION AND LIMITATIONS

Software and hardware programming. eHDL enables software network programmers to define their own hardware. For instance, accelerating Suricata took us about 1h, mostly spent in hardware synthesis. eHDL could readily generate the hardware design from the cloned Suricata's GIT repository, in few seconds, giving us an FPGA NIC-accelerated Suricata appliance. Currently, technology vendors need months of development to achieve the same, e.g., [31]. Here, it is worthy of notice that even the interface with the host system stays unchanged: programmer receive network data using sockets (e.g, of the type AF_XDP for fast packet processing); and they can update the memory hosted on the NIC using the standard eBPF map interface towards user space.

For what concerns the interactions with the host system, we did not find cases in which the eBPF map host accesses significantly affect data plane performance. We observe two most common kinds of interactions between the host system and eBPF maps:

- The data plane writes on the maps, the host only reads them. This is typical of monitoring applications (e.g., to fetch statistics). The map reads occur at a much lower frequency than accesses required for packet processing; furthermore, FPGAs can easily implement memories with multiple read ports to fully parallelize reads.
- The host writes maps, the data plane only reads them. This is the case of applications such as Access Control Lists and Routing, which use maps that host policies (e.g., a forwarding table). Also in these applications, the write access frequency is orders of magnitude lower than the map read frequency required by data plane processing, and thus the performance impact is negligible.

Reasoning for pipeline packet-streaming. The level of optimization that we manage to achieve when analyzing the eBPF program allows us to deploy the pipeline architecture as discussed. Nonetheless, FPGA experts might find the generated pipelines similar to an instance of a complex shifter register, with significant FPGA resource requirements. In practice, as we discussed in Section 5.4, our optimization is able to prune the number of used registers significantly. Furthermore, the target architecture (the FPGA) has a huge number of registers and is able to use a single LUT as a shift register, further reducing resource usage. If resources are still a concern, a simple technique to reduce them would be to indirectly index several FPGA block RAMs, reducing the number of registers used for the shifter register. However, after the first tests, we realized that also without these solutions, eHDL generated relatively resource-efficient pipelines. In addition, it is interesting to highlight that one of the targets of our work is to face the worst case of minimum packet size. This is a classic worst-case in network applications. For this scenario, we need to access the packet data and state of multiple packets. With our solution eHDL is able to process one minimum packet per pipeline stage (more than 100 packets in parallel in the longer pipelines). We suspect that indirect access to data would instead fragment the memories so much that the FPGA synthesis would map them into registers, thereby obtaining a similar (or worse) final result, i.e., using many of the FPGA registers.

Comparison with SIMD architectures. One alternative design option to scale throughput could be a parallel, SIMD-style architecture, such as a multi-core version of hXDP. Nonetheless, this approach has several drawbacks. SIMD processing of several packets in parallel requires processed packets to follow the same control path. In general, this requires batching packets that can be processed in parallel. Even assuming that a solution to implement proper batching at packet processing speeds is available, batching still adds processing latency and increases buffer size (there would be a need to keep multiple batches to hold the packets requiring the same type of processing). These are major drawbacks in high performance network devices. Furthermore, in order to achieve a 10x throughput improvement using a design built with a multi-core hXDP approach, ignoring cross-core memory synchronization issues for simplicity, and assuming a perfect linear scaling, we would need 10 hXDP cores. This would translate to about 10x larger hardware resource requirements than eHDL.

Limitations and open issues. eHDL provides the flexibility of eBPF and the performance of specialized hardware, but there are still limitations and open issues.

Target Platform. eHDL defines a hardware design at the HDL level. This means that the design is readily suitable for existing FPGA targets. It is still unclear if defining hardware features for NICs using eBPF would be a valuable tool also for chip designers, in addition to FPGA users.

Accessing host's maps. While eHDL designs are able to access host's memory and maps located in the Linux environment, doing so comes with a performance penalty due to data movements between NIC and host system, e.g., PCIe transactions. Therefore, while programs run unchanged on e.g., an FPGA NICs, operators need to be aware of the accessed maps location, when performing program deployments. Automatically managing and orchestrating the runtime of programs on heterogeneous architecture is an open research problem.

Program changes. eBPF programs are usually loaded at runtime. Even when targeting an FPGA platform, which supports hardware reconfiguration, the synthesis of an eHDL-generated pipeline usually takes few hours, due to the hardware synthesis process. Thus, development cycles for network functions, as well as servicing procedures need to change. Even if a synthesized pipeline is already available for an FPGA target, loading it requires putting the FPGA NIC out of service, to re-flash it. Considering the low resource requirements of eHDL pipelines, we plan to explore the use of dynamic partial reconfiguration to enable dynamic loading of programs in future.

7 CONCLUSION

eHDL generates specialized hardware architectures to run (unmodified) eBPF programs provided as input. The generated hardware can run such programs at line rate on current 100Gbps FPGA NICs (and up to 250Mpps), with an end-to-end forwarding latency of about $1\mu s$. Unlike general high-level synthesis tools, eHDL does not require any hardware expertise. Compared to network-specific high-level synthesis tools, it is instead capable of handling more expressive stateful programs. Given that eHDL designs can already forward higher packet rates than the maximum supported on a 100Gbps port (250Mpps vs 150Mpps), and considering the small hardware resource requirements and the pipelined processing model, we speculate that a similar design is also suitable to address the needs of NICs with higher port speeds (e.g., >200Gbps). We believe eHDL is a first step towards enabling network application developers to define their own NIC's hardware functions, thereby scaling per-node performance for the upcoming faster network port speeds.

ACKNOWLEDGMENTS

This work has been partially funded by the European Commission in the frame of the Horizon 2020 project 5GMED (grant #951947) and MARSAL (grant #101017171), and project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU, and the ECSEL Joint Undertaking, under grant agreement n. 876967 ("BRAINE").

A APPENDIX

A.1 Modeling of Throughput Degradation Due to Flushing

We can analytically compute the throughput achievable when flushing occurs considering two parameters: (i) the number of stages K that are flushed, which in the current implementation are the stages from the beginning of the pipeline up to the stage where a data hazard occurred, and (ii) the number of stages L between the write and the read stages. It must be noted that in the current implementation K has an additional overhead of 4 clock cycles used to reload the pipeline after a flush.

If we have N flows, and we suppose that the flows are uniformly distributed it is possible to compute the flushing probability as the probability that two packets of the same flow are inside one of the L stages. This is the standard birthday paradox, so the flushing probability can be roughly approximated as:

$$P_f^u = 1 - e^{-\frac{L^2}{2N}} \tag{1}$$

A more realistic setting uses a Zipfian distribution of the packets, in which $i-th$ flow has a frequency $f_i \propto 1/i$. Since

$$\sum_{i=1}^{N} 1/i \approx ln(N)$$

we can define the probability of getting the $i-th$ flow as $P_i^Z = \frac{1}{i \cdot ln(N)}$, where the $ln(N)$ factor is used to normalize to 1 the sum of all the P_i^Z.

The flushing probability $P_f^Z(i)$ caused by the $i-th$ flow can be approximated as the probability to have at least two occurrences of the $i-th$ flow in L trials. This can be computed as

$$P_f^Z(i) \approx \frac{L(L-1)}{2}(P_i^Z)^2(1-P_i^Z)^{L-2}$$

Thus the overall P_f^Z is

$$P_f^Z = \sum_{i=i}^{N} P_f^Z(i)$$

Considering a theoretical throughput of one packet for clock cycles we can achieve up to $T = 250Mpps$ when no flushing occurs. When the flushing occurs the throughput decreases to $T_f = T/K$. Thus the overall throughput of the pipeline T_p is

$$T_p = \frac{T}{(1-P_f)+KP_f} \tag{2}$$

we can retrieve the maximum number of stages K_{max} that can be flushed maintaining a minimum pipeline throughput as:

$$K_{max} = \frac{T/T_p - (1-P_f)}{P_f} \tag{3}$$

In all the practical settings the number of flows is at least 50K and L is limited. We reported in table 3 the number K of stages to be flushed and the number of stages L between the read and the write stage for the use cases discussed in §5 for the performance assessment. It is possible to see that L is usually in the range 1-3. It is worth to notice that for many of the use case in the table, the atomic primitive could be also used to avoid flushing. The only exception in the value of L is the DNAT use case in which the delay between the read and write stage is significant. This is due to the nature of the network function that need to execute a complex sequence of instructions when a miss occurs in the first read operation. This corresponds to the binding of a new flow in the connection table.

Table 3: Pipeline throughput T_p for different use cases supposing 50K flows with a Zipfian distribution.

Program	K	L	T_p
Simple firewall	N/A	N/A	N/A
Tunnel	109	2	120 Mpps
Router	41	2	178 Mpps
DNAT	33	51	N/A
Suricata	59	3	91 Mpps
Leaky_bucket	39	5	52 Mpps

It must be noticed that the impact of the flushing on this case only happens when a new flow arrives, while in the standard condition no write operations are done into the eBPF maps, and thus no flushing will occur in the pipeline. Excluding this case, we can compute the K_{max} that can sustain 148 Mpps (corresponding to the saturation of the 100 Gbps link) with different values of L, under the assumption of Zipfian distribution, as reported in table 4. We can also report the maximum T_p achievable by the specific program in the last column of table 3.

Table 4: Maximum number of stages that can sustain maximum throughput for different values of L, under the assumption of Zipfian distribution

L	P_f^Z	K_{max}
2	1%	61
3	3%	21
4	6%	11
5	10%	7

As can be seen, there are several use cases that theoretically cannot sustain the Zipfian traffic condition with minimal-size packets. Furthermore, it is possible to reduce the number K of stages to be flushed by inserting an elastic buffer in the pipeline. This buffer should store the pipeline registers when the flushing signal is raised, without propagating the flushing up to the beginning of the pipeline. This reduces the throughput penalty to the number of stages between the elastic buffer and the write stage. However, the experiments that we presented in section §5 shows that the actual degradation of the throughput is much less significant than the one foreseen from this model. Thus we did not use the elastic buffer to reduce the number of stages in the current implementation of the eHDL toolchain.

We also remark that for several of the above-reported use cases, we in fact apply the atomic operations block, which prevents the need to flush the pipeline.

A.2 Flushing with Side Effects

eHDL flushes a pipeline in case of a RAW hazard. If a program accesses multiple maps during its execution, there is a risk that a pipeline that already modified values in earlier maps is flushed while accessing a later map, thereby leading to a wrong system state. This problem is addressed by constraining the portion of the flushed pipeline. That is, when a pipeline includes multiple map accesses, elastic buffers (similar to the packets input queue) are added in between pipeline stages. Flushing happens only starting from the last buffer, instead of propagating until the start of the pipeline. This way, writing to earlier maps is not repeated if there is a need to flush later stages of the pipeline.

A.3 Instruction Level Parallelism

In Section 3.3, we mention that a key feature of eHDL is the ability to leverage ILP to generate faster pipelines, without the need to compromise on hardware resources. In fact, eHDL's area usage is directly proportional to the number of pipeline stages it has to implement. Thus, a reduction in the number of stages is crucial to the minimization of used hardware resources: each instruction processed in parallel with other ones shortens the pipeline by one stage. Table 5 reports ILP values associated with the use cases discussed in the paper.

```
void lookup_and_incr(
  stream<ap_uint<16>>  &key) {
  #pragma HLS PIPELINE II=1
  #pragma HLS INLINE off
  static ap_uint<64>    curr_value;
  static ap_uint<64>    updating_value;
  static enum aState
        {map_IDLE, map_UPDATE} mapState;
  static hashTableEntry hashTable[256];
  hashTableEntry currEntry;
  hashTableEntry updatingEntry;
  #pragma HLS BIND_STORAGE variable=
      hashTable type=RAM_1P impl=BRAM
  #pragma HLS DEPENDENCE variable=hashTable
        inter false

  switch(mapState) {
  case map_IDLE:
   if (!key.empty()) {
    key.read(currEntry.key);
    currEntry.value =
        hashTable[currEntry.key].value;
    mapState = map_UPDATE;}
   break;
  case map_UPDATE:
   updatingEntry.value=currEntry.value + 1;
   updatingEntry.key  =currEntry.key;
   hashTable[updatingEntry.key] =
        updatingEntry;
   mapState = map_IDLE;
   break;
  }
}
```

Listing 3: Lookup function for the program in Vitis-HLS

```
void toy_HLS(
  stream<axiWord>&  ethStreamIn,
  stream<axiWord>&  ethStreamOut,
  stream<axiWord>&  pcieStreamOut){
  #pragma HLS INTERFACE ap_ctrl_none
        port=return
  #pragma HLS DATAFLOW
  #pragma HLS interface
        mode=axis port=ethStreamIn
  #pragma HLS interface
        mode=axis port=ethStreamOut
  #pragma HLS interface
        mode=axis port=pcieStreamOut
  static stream<ap_uint<16> > key;
  #pragma HLS STREAM variable=key depth=256
  #pragma HLS aggregate variable=key
  parse_key(ethStreamIn, ethStreamOut,
            pcieStreamOut, key);
  lookup_and_incr(key);
}
```

Listing 4: Main body for the program in Vitis-HLS

```
void parse_key(
    stream<axiWord>&      ethStreamIn,
    stream<axiWord>&      ethStreamOut,
    stream<axiWord>&      pcieStreamOut,
    stream<ap_uint<16>>&  keyStream){
#pragma HLS PIPELINE II=1
#pragma HLS INLINE off
static uint16_t wordCount, key;
axiWord currWord;
packet pkt;
 if (!ethStreamIn.empty()) {
  ethStreamIn.read(currWord);
  ethStreamOut.write(currWord);
  switch(wordCount) {
  case 0:
   pkt.dstMac = currWord.data(47, 0);
   pkt.srcMac(15,0)=currWord.data(63,48);
  break;
  case 1:
   pkt.srcMac(47,16)=currWord.data(31,0);
   pkt.ethType = currWord.data(47, 32);
   pkt.hwType = currWord.data(63, 48);
  break;
  default: break;}

  if (currWord.last == 1) {
   key = 0;
   if (pkt.ethType == ETH_P_IP)
    key = 1;
   else if (pkt.ethType == ETH_P_IPV6)
    key = 2;
   else if (pkt.ethType == ETH_P_ARP)
    key = 3;
   keyStream.write(key);
   wordCount = 0;
  }else
   wordCount++;   }
}
```

Listing 5: Parse function for the program in Vitis-HLS

Table 5: Instruction level parallelism values achievable for the eBPF usecases tested.

	max ILP	avg ILP
Simple Firewall	3	1.48
Tunnel	15	2.37
Router	5	1.54
DNAT	7	1.67
Suricata	3	1.42

As previously highlighted in Section 3.3, eHDL does not face the design trade-offs of fixed processors architectures. For instance, each stage can grow to an arbitrary amount of instruction parallelism. For example in some cases, like Tunnel, there is at least one stage with 15 instructions executed in parallel. This is the outcome of a looser set of control and data dependencies in the program. When considering all the stages, the average ILP for the implemented eBPF programs is instead between 1.5 and 2.5. This is in line with the numbers reported by previous work [5].

A.4 C++ Program for Vitis HLS

We report the C++ source code we developed to implement with Vitis HLS an equivalent version of the program we used as running example we used throughout the paper. As it can be seen by the amount of code and exotic data structure and `pragmas`, state-of-the-art HLS tools expect the programmers to be in fact a hardware expert. The program is split in three parts for readibility: Listing 4 contains the program "entry point"; Listing 5 implements packet parsing; and Listing 3 implements the "map" lookup and value update function.

REFERENCES

[1] 2019. The CAIDA Anonymized Internet Traces Dataset. https://www.caida.org/catalog/datasets/passive_dataset/.

[2] Chris Arges. 2023. How We Used eBPF to Build Programmable Packet Filtering in Magic Firewall. https://blog.cloudflare.com/programmable-packet-filtering-with-magic-firewall/.

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[4] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 99–110.

[5] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2020. hXDP: Efficient Software Packet Processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 973–990.

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

[7] Sean Choi, Muhammad Shahbaz, Balaji Prabhakar, and Mendel Rosenblum. 2020. λ-nic: Interactive serverless compute on programmable smartnics. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 67–77.

[8] Cilium. 2023. Cilium - Linux Native, API-Aware Networking and Security for Containers. https://cilium.io/.

[9] Philippe Coussy, Daniel D Gajski, Michael Meredith, and Andres Takach. 2009. An introduction to high-level synthesis. *IEEE Design & Test of Computers* 26, 4 (2009), 8–17.

[10] DPDK. 2023. Data Plane Development Kit. https://www.dpdk.org//.

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

[12] Jiawei Fei, Chen-Yu Ho, Atal N. Sahu, Marco Canini, and Amedeo Sapio. 2021. Efficient Sparse Collective Communication and Its Application to Accelerate Distributed Deep Learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 676–691. https://doi.org/10.1145/3452296.3472904

[13] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 51–66.

[14] Romain Fontugne, Pierre Borgnat, Patrice Abry, and Kensuke Fukuda. 2010. MAWILab: Combining Diverse Anomaly Detectors for Automated Anomaly Labeling and Performance Benchmarking. In *ACM CoNEXT '10*. Philadelphia, PA, 12 pages.

[15] Alex Forencich, Alex C Snoeren, George Porter, and George Papen. 2020. Corundum: an open-source 100-Gbps NIC. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 38–46.

[16] Fungible. 2021. Fungible F1 Data Processing Unit. https://www.fungible.com/wp-content/uploads/2021/09/PB0028.02.12020914-Fungible-F1-Data-Processing-Unit.pdf.

[17] Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, and Srinivas Narayana. 2019. Autogenerating fast packet-processing code using program synthesis. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*. 150–160.

[18] Hubble github repository. 2023. https://github.com/cilium/hubble.

[19] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. 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* (Heraklion, Greece) *(CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 54–66. https://doi.org/10.1145/3281411.3281443

[20] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. 2019. The p4-> netfpga workflow for line-rate packet processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 1–9.

[21] Intel. 2023. Intel Infrastructure Processing Unit (Intel IPU) and SmartNICs. https://www.intel.com/content/www/us/en/products/network-io/smartnic.html.

[22] Jakub Kicinski and Nicolaas Viljoen. 2016. eBPF Hardware Offload to SmartNICs: cls_bpf and XDP. *Proceedings of netdev* 1 (2016).

[23] Sakari Lahti, Panu Sjövall, Jarno Vanne, and Timo D Hämäläinen. 2018. Are we there yet? A study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (2018), 898–911.

[24] Leonardo Linguaglossa, Stanislav Lange, Salvatore Pontarelli, Gábor Rétvári, Dario Rossi, Thomas Zinner, Roberto Bifulco, Michael Jarschel, and Giuseppe Bianchi. 2019. Survey of Performance Acceleration Techniques for Network Function Virtualization. *Proc. IEEE* 107, 4 (2019), 746–764. https://doi.org/10.1109/JPROC.2019.2896848

[25] Devon Loehr and David Walker. 2022. Safe, Modular Packet Pipeline Programming. In *Symposium on Principles of Programming Languages*, Vol. 6.

[26] Thomas Luinaud, Jeferson Santiago da Silva, JM Pierre Langlois, and Yvon Savaria. 2021. Design Principles for Packet Deparsers on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 280–286.

[27] Scott A Mahlke, Richard E Hank, James E McCormick, David I August, and Wen-Mei W Hwu. 1995. A comparison of full and partial predicated execution support for ILP processors. In *Proceedings of the 22nd annual international symposium on Computer architecture*. 138–150.

[28] Marvell. 2021. Marvell OCTEON 10 DPU Platform. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-octeon-10-dpu-platform-product-brief.pdf.

[29] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/3098822.3098824

[30] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. 2015. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (2015), 1591–1604.

[31] Napatech. 2023. Scaling Suricata Performance to 100 Gbps With Napatech SmartNICs. https://www.napatech.com/support/resources/solution-descriptions/scaling-suricata-performance-to-100-gbps-with-napatech-smartnics/.

[32] NEC. 2020. Building an Open vRAN Ecosystem White Paper. https://www.nec.com/en/global/solutions/5g/index.html.

[33] Nvidia. 2021. Nvidia Bluefield-2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf.

[34] M Akif Özkan, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, Roland Leißa, Sebastian Hack, Jürgen Teich, and Frank Hannig. 2020. AnyHLS: high-level synthesis with partial evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3202–3214.

[35] Pensando. 2022. Distributed Services Card. https://www.amd.com/system/files/documents/pensando-dsc-200-product-brief.pdf.

[36] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. 2015. The design and implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 117–130.

[37] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for {NIC-Accelerated} Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 663–679.

[38] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, et al. 2019. Flowblaze: Stateful packet processing in hardware. In

16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). 531–548.

[39] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. 2022. Re-architecting Traffic Analysis with Neural Network Interface Cards. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 513–533.

[40] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 15–28.

[41] Suricata. 2023. Suricata IDS Website. https://suricata.io/.

[42] tomshardware.com. 2022. AMD Acquires Pensando Data Processing Units in a $1.9 Billion Deal. https://www.tomshardware.com/news/amd-acquires-pensando-data-processing-units.

[43] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4FPGA: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*. 122–135.

[44] Xilinx. 2017. P4-SDNet User Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1252-p4-sdnet.pdf.

[45] Xilinx. 2021. Alveo SN1000 SmartNICs. https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/xilinx-alveo-sn1000-product-brief.pdf.

[46] Xilinx. 2022. Introduction to Vitis HLS. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Introduction-to-Vitis-HLS.

[47] Noa Zilberman, Yury Audzevich, G Adam Covington, and Andrew W Moore. 2014. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE micro* 34, 5 (2014), 32–41.

[48] Noa Zilberman, Gabi Bracha, and Golan Schzukin. 2019. Stardust: Divide and Conquer in the Data Center Network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 141–160. https://www.usenix.org/conference/nsdi19/presentation/zilberman

Received 2022-10-20; accepted 2023-01-19