eBPFlow: A Hardware/Software Platform to Seamlessly Offload Network Functions Leveraging eBPF

Racyus D. G. Pacífico, Lucas F. S. Duarte, Luiz F. M. Vieira, Barath Raghavan, José A. M. Nacif, *Member, IEEE*, and Marcos A. M. Vieira

***Abstract*— NFV and SDN enable flexibility and programmability at the data plane. In addition, offloading packet processing to a hardware saves processing resources to compute other workloads. However, fulfilling requirements such as high throughput and low latency with a flexible and programmable data plane is challenging. This paper introduces eBPFlow, a platform for seamlessly accelerating network computation. It builds upon eBPF. eBPFlow combines flexibility and programmability in software with high performance using an FPGA. We implemented our system on the NetFPGA SUME, performing tests on a physical testbed. We built a range of NFs. Our results show that the eBPFlow supports offloading of NFs with throughput at the line rate, latency between 20 μs and 40 μs, communication with host, and consumption of 22 W. Moreover, eBPFlow processes 12.05 Mpps more than the kernel. eBPFlow has a throughput of 2.59 Gbps higher than the hXDP, a system similar to eBPFlow.**

***Index Terms*— Networking functions virtualization, programmable data plane, eBPF, NetFPGA.**

I. Introduction

NETWORK Function Virtualization (NFV) and Software-Defined Networking (SDN) provide flexibility and programmability on the network data plane. Combining these technologies improves manageability, reliability, and agility, enabling network operators to adapt to both upgrades and service demands. However, NFVs processing typically occurs in software on virtual machines or containers of commodity servers. Such software dataplanes, while much faster today than a decade ago, struggle to support today's traffic demands.

Numerous recent studies have aimed to mitigate the poor performance of software, while retaining their flexibility, by offloading network functions (NFs) to hardware accelerators such as programmable switches and SmartNICs [1]. NFs can be partially or entirely offloaded and accelerated. Programmable data planes provide programmability and flexibility to implement different tasks on network devices, enabling adaptability for new headers and protocols. Also, they support NF offloading, improving processing performance. However, each offload platform brings with it major limitations on generality (e.g., P4 can only support a narrow range of types of NF) and expressiveness. Thus to date most efforts have focused on bespoke implementations for a specific offload platform rather than developing a fast, general-purpose approach to NF offload [2], [3].

Moreover, there are challenges in programming FPGA for NF, such as:

Programmability and flexibility: FPGAs are hardware platforms that combine flexibility in software with high processing power. Due to these features, FPGAs are attractive platforms to accelerate packet processing and offload NFs. Moreover, they are reprogrammable with power efficiency. On the other hand, hardware programming occurs through low-level hardware description languages (HDLs) such as Verilog and VHDL, which do not offer high productivity rates [4].

Achieve high-performance: Hardware to offload NFs and accelerate packet processing must support many stateless and stateful functions (e.g., tunneling, forwarding, traffic shaping, monitoring, access control list (ACL), firewall, and DDoS protection) with a throughput of 40-200 Gbps. Moreover, it should minimize processing overheads and performance bottlenecks. All these points directly affect the hardware performance and quality of services.

To address these challenges, we propose eBPFlow, a platform that supports offloading NFs using the standard, general-purpose eBPF (extended Berkeley Packet Filter) instruction set [5] already used widely in the Linux kernel. eBPF specifies a bytecode machine and an instruction set that we leverage to program general-purpose NFs on the data plane. eBPFlow is a platform for seamlessly accelerating network computation to deploy building upon eBPF. Moreover, it combines flexibility and programmability in software with high performance in hardware using an FPGA (Field Programmable Gate Array).

Furthermore, eBPFlow is protocol-independent, allowing the utilization of new dynamically defined fields and protocols without recompiling or restarting the device when the user changes the packet processing algorithm on the data plane at runtime. eBPFlow supports all network hardware requirements to offload and accelerate NFs, such as similar integration,

Manuscript received 14 April 2022; revised 24 April 2023 and 20 July 2023; accepted 5 September 2023; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor J. Llorca. *(Corresponding author: Racyus D. G. Pacífico.)*

Racyus D. G. Pacífico, Luiz F. M. Vieira, and Marcos A. M. Vieira are with the Department of Computer Science, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte 31270-901, Brazil (e-mail: racyus@dcc.ufmg.br).

Lucas F. S. Duarte and José A. M. Nacif are with the Department of Computer Science, Universidade Federal de Viçosa (UFV), Viçosa 36570-900, Brazil.

Barath Raghavan is with the Department of Computer Science, University of Southern California (USC), Los Angeles, CA 90007 USA.

Digital Object Identifier 10.1109/TNET.2023.3318251

performance, programmability, flexibility, lookup and pattern matching, forwarding, traffic shaping and control, serviceability, and data manipulation. eBPFlow runs on the NetFPGA SUME 40 Gbps platform [6]. The tests were performed in a physical testbed, demonstrating the eBPFlow performance to offload stateless and stateful NFs and accelerate processing. We present the feasibility of building NFs such as LPM forwarding, Stateful firewall, DDoS mitigation, and Deep Packet Inspection (DPI).

The main contributions of this work are the following. First, offloading network functions and accelerating packet processing by leveraging eBPF and integrating existing eBPF environments and projects. Second, eBPFlow allows users with little hardware expertise to develop functions that operate on packet headers and payload, L2-L7 layers of the network stack with high throughput and low latency. Third, logic design and hardware implementation of eBPFlow, built on top of the NetFPGA SUME [6] with three parallelism types: instructions parallelism with a multi-core hardware design containing 5-stages pipeline; parallelism per port through a cores group reserved per port; and parallelism on the packet forwarding through an output crossbar coupled on the data path. Finally, eBPFlow enables the programming of stateful and stateless NFs and the use of dynamically defined new fields and protocols at runtime.

This brief history of recent programmable data planes illustrates the industry's trend of adopting eBPF. Pacifico et al. [7] proposed a simplified version of the eBPFlow on NetFPGA SUME. In this version, the system contains four eBPF engines shared between all the ports. Each eBPF engine has a pipeline with 5-stages, providing parallelism of instructions. However, this system does not support parallelism in forwarding packets and per-port with a number of exclusive cores, harming the system's performance due to lost packets and processing overhead. Here, we extended this work by providing new types of parallelism (per port and forwarding of packets), increasing the number of eBPF cores, and adding an output crossbar. We have also realized new experiments to evaluate and compare the system with similar systems. Netronome [8] provides a SmartNIC that includes programming capabilities with eBPF instructions, showing the trend towards programmable data planes with eBPF. But, to program the SmartNIC, the code has to pass a verifier that disables back-edge jump (e.g., for, while loops), so the SmartNIC can not execute NFs that compute on the packet payload (e.g., DPI). Moreover, Netronome is firmware and kernel-dependent, making it challenging to manage the network; it has a very low port density (only 2 ports); it does not provide specific hardware modules, such as CAM or TCAM to handle stateful NFs. Finally, hXDP executes XDP code in hardware. Besides eBPF ISA, hXDP also provides new instructions. But, it does not support offloading of NFs in runtime. The eBPFlow is compatible with the eBPF standard [9]. Moreover, it presents more return codes of the design in hardware. This does not harm the compatibility with other eBPF systems.

The remainder of this paper is organized as follows. In §II, we introduce an overview for understanding the eBPFlow. In §III, we describe the eBPFlow design. In §IV, we present the implementation details of eBPFlow built on top of the NetFPGA SUME platform. In §V, we describe the network functions evaluated on our system. In §VI, we show the evaluation and results in a realistic environment. In §VII, we describe and compare the related work. Finally, in §VIII, we present the conclusion.

II. Overview

This section presents an overview for understanding the eBPFlow.

A. eBPF

eBPF is a general-purpose soft 64-bit processor available on Linux kernel since version 3.15. It allows fast processing of packets at runtime inside the kernel and provides programmability and flexibility on packet computing. Users can compile eBPF programs to bytecode before loading it on the kernel. Languages such as C and P4 support this technology. More details about eBPF are available in [5], a complete course. High-level languages are used to write code to the data plane and compile it into the eBPF instruction set. Since version 3.7, the LLVM compiler collection has a backend for the eBPF platform, allowing programming in this subset of C and generating executable code in eBPF format. Many projects use eBPF, e.g., Facebook built a layer 4 load balancing forwarding plane using eBPF to provide fast packet processing in-kernel. Moreover, problems such as interdependence, distribution, and heterogeneous hardware can be solved due to the features and environments available in this technology.

B. NetFPGA

The NetFPGA project [10] is an open-source project that leverages research and development of new networking applications using a SmartNIC based on FPGA. It provides a platform composed of software and hardware. All platforms' infrastructure aims to simplify development tasks such as design, simulation, and testing of high-speed networking applications in hardware. The NetFPGA's development environment allows the creation of new designs reusing the base code of reference projects (e.g., NIC, Switch, and IPv4 Router). In addition, it has support from a broad research community. We chose the NetFPGA to demonstrate the system due to the platform's benefits. Moreover, the NetFPGA platform allows bypass challenges of the system's design, creating new circuits to provide parallelism and using the resources available in the platform, such as IP cores, FIFOs, and memories, to optimize the performance of the system.

III. eBPFlow Design

Figure 1 gives an overview of the eBPFlow's design and implementation built on top of the NetFPGA SUME [6] platform. The system has two components: data plane and userspace tools.

The data plane contains sixteen processing cores divided into groups composed of four eBPF engines, with each group

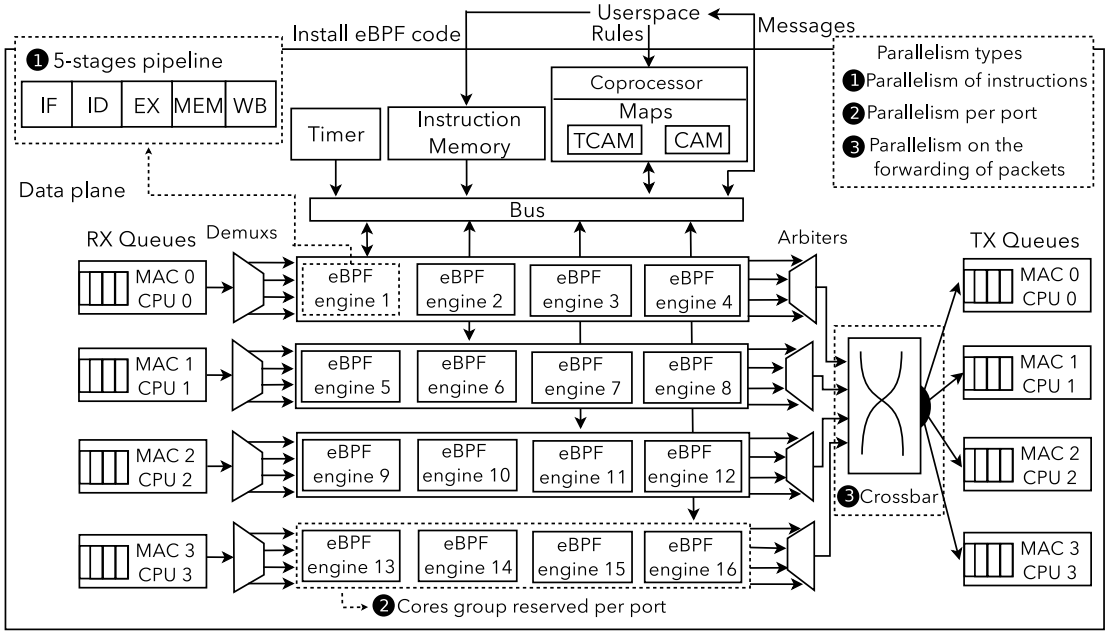


Fig. 1. eBPFlow design.

responsible for packet processing and forwarding for each RX and TX queue. We chose the number of eBPF engines (four) per group based on the design's consumed resource (logic cells) and the maximum frequency obtained after the synthesis. All groups share an instruction memory, a timer, a coprocessor, and an output crossbar. The instruction memory includes a system with a double buffer that changes programs without stopping the processing. The timer allows measurement of network performance (for example, through EWMA, latency, and jitter) when storing the timestamp of packets on metadata. The coprocessor works as eBPF maps in hardware using TCAM/CAM memories to store pairs <key, value>. The output crossbar provides parallelism in the forwarding of packets, improving the throughput and latency of the system. Moreover, each group has one demux and output arbiter reserved per RX and TX queue, which allows the system to receive and send packets exchanging cores in runtime. We divided the eBPF engines into groups by queue to provide per-port parallelism on packet processing. Moreover, we added an output crossbar to receive the processed packets per each group to all output queues, providing parallelism on forwarding.

On eBPFlow, four parallel engines per port can cause packet reordering in a single flow. We do not treat the packet reordering on the system, leaving the TCP protocol responsible for this task because it uses the flow control mechanism to bypass the packet reordering in a single flow.

The userspace includes a controller that opens a socket connection TCP/IP to the device and applications created at the user level as a loader to compile/load programs and handle maps, an eBPF disassembler to convert binary code to eBPF instructions, a software emulator to debug, and a CLI application to interact with eBPF engines.

A. How Does eBPFlow Work?

The packet processing on eBPFlow begins with the user-generated eBPF instructions via C eBPF or P4 code on userspace. Once generated, the instructions take their course from userspace to the data plane, where the system loads them into the instruction memory. The communication between userspace and data plane occurs through userspace tools, loader, and PCIe bus. All processing on the platform occurs without the user knowing specific low-level commands or having experience with hardware targets.

The processing on the data plane begins with the packet's arrival in an Rx queue. RX's queue demux forwards the packet to the current eBPF engine according to the register value that controls which eBPF engine has access priority. The access priority algorithm between eBPF engines is a standard Round Robin (RR) algorithm. If an eBPF engine cannot receive a packet, the system updates the register to the next eBPF engine. Each eBPF engine waits for the first packet word to arrive. In the next step, eBPFlow processes and forwards the packet to the output arbiter. It selects the current eBPF engine packet that terminates the processing and sends it to the output crossbar. The output arbiter chooses the priority of access of the current eBPF equal to the demux using the standard RR algorithm. The output crossbar receives packets of the output arbiters simultaneously, it parallelizes the forwarding of packets, and decides which TX queue will store the packet. In its turn, the TX queue sends or drops the packet according to the value stored in the eBPF r0 register. Each engine has one action module. Thus, for 16 eBPF engines, we have 16 action modules. The action module updates the packet's metadata for the TX queue to process it. The TX queue is responsible for dropping the packet. We present more details about eBPFlow in Section IV.

B. How Does eBPFlow Provide Flexibility and Programmability of the Data Plane?

The eBPFlow is not tied to specific network protocols, enabling programmers to perform runtime parse, match, and action operations dynamically. On eBPFlow, programmers can

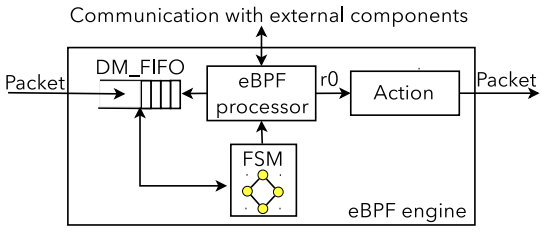


Fig. 2. eBPF engine design.

change how the system processes packets after the design is synthesized and loaded on hardware. This feature allows the system to provide the flexibility of the data plane, defining the packet processing logic in two ways: (i) Reconfigurable in the field; and (ii) processing protocol-independent packets. The combination of these functionalities allows programmers to insert new fields and protocols. The eBPF technology is responsible for the system's flexibility using eBPF instructions generated from programs in the C language.

eBPFlow supports the standard eBPF ISA, allowing similar integration with other existing eBPF environments and projects on the network. In addition, the eBPFlow provides programmability, enabling programmers to describe packet processing logic independent of the specifics of the underlying hardware. This feature becomes the target-independent eBPFlow. Programmers only need to know the eBPF technology to use the eBPFlow. It is possible due to a combination of software and hardware technologies implemented on the userspace and data plane of the system. For example, hXDP [11] is similar to the eBPFlow, and it executes XDP code. However, hXDP is incompatible with eBPF because of the addition of new instructions, thus changing the standard eBPF ISA.

C. eBPF Engine

Each eBPF engine comprises four hardware modules: a Data memory FIFO (DM_FIFO), the eBPF processor, a Finite State Machine (FSM), and an action module. Figure 2 shows the eBPF engine design. The DM_FIFO stores packets on the fly, working as data memory and FIFO with no extra transfer. The eBPF processor is responsible for performing the parse, matching, and actions using instructions stored in the instruction memory. Also, the processor communicates with the control plane through a socket TCP/IP. The FSM controls the whole operation of packet processing. It removes the packet from the DM_FIFO of the module, starts executing the eBPF instructions, and forwards the packet to the next module (action packet) when the last instruction (`exit`) of eBPF finishes executing. The action module forwards or discards the packet according to the value stored in r0 after processing the eBPF instructions.

D. Metadata

The data plane receives the packet through the input interface and stores the packet in the input queue with additional information called metadata. Table I shows the metadata header. The first line indicates the byte order and size. The other lines show the stored structure. After the metadata is received, it comes the Ethernet frame. eBPF programs and any other protocol field can use the metadata header fields. The currently defined metadata is the destination port, packet size in multiples of 64-bit, source port, packet size in bytes, timestamp in nanoseconds, and seconds. The fields of packet size, in multiples of 64-bit and bytes, are included because the input queue module already provided this information.

TABLE I

METADATA: INFORMATION RETRIEVED FROM THE INPUT QUEUE OF THE STORED PACKET IN THE DATA MEMORY OF THE EBPF PROCESSOR

0			bit			255
8 bits	8 bits	8 bits	32 bits	32 bits	16 bits	152 bits
Input Port	Source Queue	Destination Queue	Time-stamp (s)	Time-stamp (ns)	Length (bytes)	Free
Ethernet Frame						
Payload						

TABLE II

ACTION PERFORMED ON THE PACKETS

Action	Code	Description
Forwarding	0 - 0xFFEF	Forward packet to a specific port.
Controller	0xFFF3	Send packet to the controller.
Drop	0xFFF0	Drop packet.
Flood	0xFFFF	Send packet to all ports except for the input port.
Host	0xFFF[2-5]	Send packet to host. (CPU Queue: 0-3).

E. Actions

The register r0 stores the return value of the eBPF processor. In addition, it determines which action the processor will execute on the packet. Table II describes the return values of eBPF and their respective actions. After eBPF finishes the computation, the packet can be: forwarded to a port, forwarded to the controller, discarded, flooded to all ports except for the input port, or sent to the host machine via PCIe bus. eBPFlow enables other dynamic actions such as modifying the packet header, packet payload, and adding or removing fields. With the packet stored in the data memory, a store instruction can modify the packet. The packet content can also be used for arithmetic and logical operations, for example, decrementing TTL or recomputing checksum.

F. Output Crossbar

On eBPFlow, the output crossbar allows connecting the eBPF engine outputs to TX queues using the output queues (OQ) module of the NetFPPGA's datapath as a buffer. Packets processed by eBPF engines are forwarded to output queues modules and TX queues via crossbar interconnect. TX queues receive the packets based on the destination queue metadata field generated by eBPF engines. This field receives the r0 value updated of the eBPF engine after the eBPF program finishes. The output crossbar works on non-blocking mode, allowing multiple simultaneous packet forwarding for different TX queues. We used an N-to-M uni-directional crossbar interconnection architecture to connect output queues to TX queues. N is the number of output queues modules, and M is

the number of TX queues. The crossbar interconnection has a size equal to $N \times M$ $(4 \times 4 = 16\ points)$.

IV. Implementation

Here, we describe the implementation details of eBPFlow. We built eBPFlow data plane in Verilog HDL on the top of the NetFPGA SUME platform and created tools on userspace to manage operations of the system [12].

Hardware Instance: FPGA enables the building of hardware logic systems. The NetFPGA SUME hardware has four SFP+ transceivers that support 10 Gbps Ethernet ports. It connects to a motherboard through a PCIe Gen 3 $\times$ 8 adapter. In addition, it contains a Xilinx Virtex-7 690T FPGA [13], which has approximately 693,120 logic cells, a 27 MiB SRAM, and a 5 ns (200 MHz) clock cycle. After synthesis, eBPFlow consumed 20.71% of the logical slices and 11.35% of the register slices on the NetFPGA SUME. The maximum frequency is 166.67 MHz (cycle of 6.172 ns).

A. eBPF Processor With Pipeline

The eBPF processor performs the parse, matching, and actions according to the user-generated C-code or P4-generated eBPF instructions. When starting the device operation, the user must load the eBPF instructions into the instruction memory to define the behavior of the data plane. Figure 3 presents the data and control paths in register transfer level (RTL), containing five data functional units (program counter, instruction memory, register file, arithmetic logic unit – ALU, and data memory) and three control units (hazard detection, forwarding, and control).

After the instruction memory returns the instruction pointed by the current program counter, eBPFlow divides the instruction into five parts: operation code, destination register address, source register address, offset, and immediate value. Each specific unit of the datapath receives part of the instruction. The control unit receives the operation code and forwards the control signals to the functional units, defining the behavior of each unit. For example, the ALU class instructions do not use the data memory, so the read and write signals from the data memory are not activated.

We design the eBPF processor with a 5-stage pipeline: instruction fetch (IF), instruction decode (ID), execute (EXE), memory (MEM), and write back (WB). IF stage gets instruction from memory and increments program counter (PC). ID stage translates opcode into control signals and reads registers from the register file. EXE stage performs ALU operation and computes jump/branch targets. MEM stage accesses data memory if needed. Finally, the WB stage updates the register file. This design follows the MIPS load-store pipeline architecture [14]. We add four pipeline registers (between the stages), the forwarding, and hazard units.

B. Data Memory (Optimized FIFO)

To avoid the overhead of copying the packet from the transfer FIFO to the processor data memory, we designed a new abstract data type called Data Memory FIFO (DM_FIFO), which works as a FIFO and as well a Data Memory. DM_FIFO enables the eBPF processor to access the packet's data without waiting for all the packets to arrive with no extra transfer and running load and store operations with high efficiency. Moreover, DM_FIFO synchronizes with the NetFPGA datapath's modules (input arbiter and output queues) and the eBPF processor. Therefore, the packet does not need to be initially stored on FIFO and forwarded to data memory to be processed by the processor.

We added two-way communication to communicate with NetFPGA's modules and eBPF processor. Thus, DM_FIFO works as a FIFO receiving and forwarding packets using the AXI4 stream interface signals. At the same time, DM_FIFO also operates as a data memory that communicates with the eBPF processor using control signals sent by the control unit of the eBPF processor's data path and load and store instructions. As a result, DM_FIFO has a capacity of 2.048 bytes (64 depth lines $\times$ 256 bits width) and can store up to 32 packets of 64 bytes.

The eBPF engine can start processing the packet even if the packet has not fully arrived yet. Inside DM_FIFO, for each word, we added a valid bit to indicate if the word contains data from the new incoming packet. Thus, DM_FIFO brings two advantages: It does not has no extra transfer and allows the eBPF engine to begin the packet processing before waiting for the entire packet to arrive.

Each engine has its own DM_FIFO. The DM_FIFO can simultaneously receive multiple packets of the datapath. But, it can only process one packet at a time. When a new packet is inserted in the DM_FIFO, and an older packet is in processing, it must wait for the processing to finish. The metadata region can not be overwritten because we use a FSM. Each FSM has its own metadata register.

1) eBPF Stack: On eBPFlow, the stack is part of the data memory on the last bytes (2,112 to 2,624). We include the stack in data memory to facilitate the eBPF engine access to the stack. eBPFlow's stack has a size of 512 bytes, the same as the Linux's kernel, following the standard of the eBPF virtual machine. Figure 4 presents the data memory structure with spaces reserved for metadata, packet, and stack. The loader is responsible for defining the value r10 on the system. Byte 2,624 (`0xa40`) is the first byte of the stack. After the generated eBPF instructions by the eBPF compiler, the loader initializes and adds one instruction with r10 value (`mov r10, 0a40`) on the eBPF program before loading the instructions on the instruction memory of the eBPFlow. If there is a local variable on the eBPF program, one of the r6 to r9 registers receives the r10 value minus the number of bytes of the local variable size to define the reserved space on the stack. With the address of the local variable defined in a register, the eBPF processor can access the local variable data on the stack through load and store instructions.

C. Instruction Memory

The eBPF instructions define the behavior of how the eBPF processor handles the packets. First, we created software registers to insert the eBPF instructions into the instruction memory through NetFPGA's register interface. The controller is responsible for sending the instructions to the data plane of

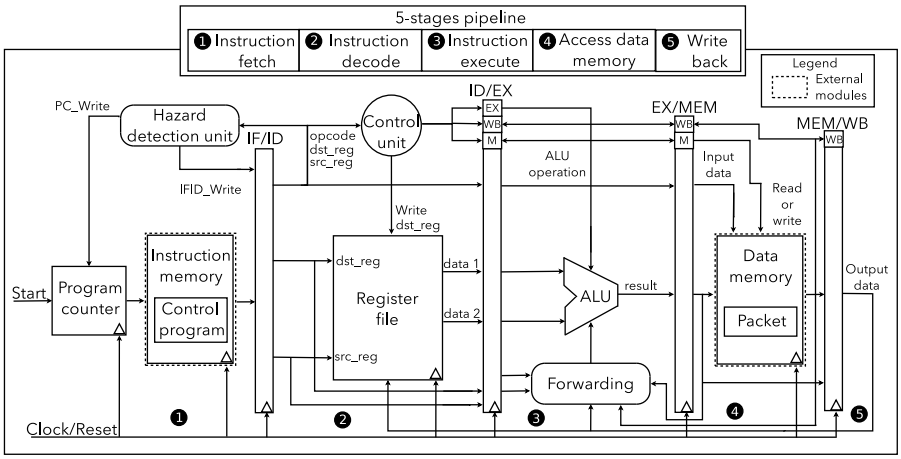


Fig. 3. Control and datapath of the eBPF processor.

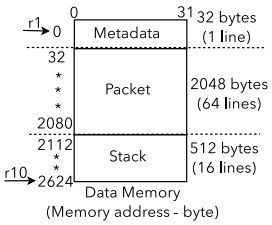


Fig. 4. Data memory division.

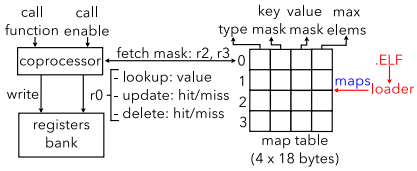


Fig. 5. Coprocessor and map table.

the eBPFlow. They are then written to the software registers using the loader and forwarded to instruction memory through the PCIe bus. Instruction memory uses a double buffer system - DBS. This system contains two memories (M_1 and M_2). Both memories never assume the same state (writing or reading) simultaneously. It means that while a memory receives eBPF program instructions, the eBPF processor reads the instructions of other memory.

As a design decision, we have put the instruction memory outside the eBPF processor to enable the connection of multiple processors using a shared instruction memory to reduce the number of used logic resources in the design. Moreover, with the increasing number of eBPF processors, it is possible to process more packets simultaneously, thus also increasing the throughput.

D. Maps

eBPF Linux kernel implementation allows for maps. A map is a generic data structure that stores different data types in the form of key-value pairs. Our design currently provides three types of maps: longest prefix matching (LPM), exact-match, and array, as hardware components of the system. For LPM, we use the Ternary Content Addressable Memory (TCAM) module combined with the BRAM (Block RAM).

Our TCAM module contains one TCAM memory with 32 lines of 64 bits. In addition, it spends 16 cycles for a write operation and only one cycle for the read operation. We use the Content Addressable Memory (CAM) module combined with the BRAM for the exact match. The CAM module contains one CAM memory with 32 lines of 64 bits. The TCAM is implemented using Xilinx SRL16e primitives [15]. It is generated using Xilinx's IP core generator *coregen* [16]. The CAM is implemented using block RAM (BRAM) instead of SRL16e. This option enables writing on CAM using two cycles instead of 16 cycles. We defined the size of 64 bits to CAM and TCAM memories to optimize the system's design to insert other functionalities on eBPFlow. The size of these memories can be extended to support keys greater than 64 bits. However, it consumes more logical resources that can harm the system's performance. Another option is to treat hash collisions using the keys of 32 and 64 bits and store 104-bit on the BRAM (Block RAM), which is already in our design. For the array map, we use the DRAM memory.

There are three functions to manipulate the maps: update, delete, and lookup. The update operation updates an item on the map. If the item does not exist, it inserts the item. The delete operation removes the item with the given key. Finally, the lookup operation searches for the key and returns an item.

The coprocessor needs to know the actual size of the data read/written on the memory map. This information is stored on fields type, key mask, value mask, and maximum number of the *maps table*, shown in Figure 5, which holds metadata about each map declared in the currently loaded program. A lookup on the map table is performed on every map operation to retrieve key and value masks used in a bitwise-AND operation with the data to clean any unwanted bits. The coprocessor also uses the map type to switch to the proper memory unit (CAM or TCAM). The r3 register stores a pointer to the item when used to operations with maps. The size of the r3 register is 64 bits based on standard eBPF architecture.

The eBPFlow supports 64 flows simultaneously using static rules through the CAM and TCAM memories. However, if the

user uses the wildcard mechanism of the TCAM memory with the operator (*), the number of flows monitored can be increased. Another mechanism to extend the number of flows is to forward the packets to userspace for offline processing via the PCIe bus. However, it is slow, decreasing the system's processing power due to the speed of the PCIe bus and context switch between hardware and userspace.

E. Call Instruction

eBPF allows invoking functions to access tables. In our design, to manipulate maps, we decided to use the TCAM, CAM, and DRAM modules. The function call inside the processor establishes communication with the coprocessor hardware module to manipulate tables. Thus, the processor communicates with the coprocessor module where there is a call instruction. This module identifies what function (lookup, update, delete) was called through the call instruction immediate opcode parameter. Registers 1 to 4 store the passed parameters on the call instruction. Register r1 indicates which hardware module to communicate (tables TCAM, CAM, DRAM, respectively). Register r2 provides the key. Register r3 stores the item of 64 bits. Register r4 has the TCAM mask item. The function return parameter is through register r0. Since the call instruction requires register values, it can also suffer from hazards in the pipeline. Therefore, the hazard unit has to stall to solve this issue.

F. Bus, Demux and Output Arbiter

On eBPFlow, we implemented demux and output arbiter modules to manage the receiving of the packet from the RX queue to the eBPF engines and forward the packet from eBPF engines to the output crossbar. Each RX/TX queue has its demux and output arbiter. These modules use the AXI-4 stream interface signals to synchronize the receiving and forwarding of the packet from/to eBPF engines. Moreover, these modules are sequential circuits that depend on AXI-4 stream interface signals value and the hardware register state to control what eBPF engine has access priority. We used the Round Robin algorithm to schedule between eBPF engines using a finite state machine that controls the hardware register responsible for access priority between eBPF engines.

G. User Space

It has a controller, a loader, and tools created at the user level. We implemented it all in the Python language.

Controller: It opens a socket connection TCP/IP to the device to exchange the messages. After establishing the connection, the operator can transmit the eBPF program already compiled as bytecode. Finally, the controller installs the bytecode in the hardware at runtime.

Loader: It is responsible for the following operations: loading code to the processors, appending two instructions, handling maps, and interacting with the processor register interface. *Loader* specially designed for the eBPF processor. At the beginning of every eBPF program, registers r1 and r10 must be initialized with two pointers: one to the packet and one to the stack's top. Since these are specific to the runtime environment (here, the processor), such initialization is not part of the code generated by *clang* compiler. To handle maps, the compiler adds map information to the eBPF ELF file as a relocation section, which needs to be processed before code execution. *Loader* adjusts all map *call* instructions with their corresponding map values according to the relocation table in the ELF file. Finally, the *loader* interacts with the system through the register interface of the hardware, allowing to update and query the content of maps from the user space at run-time independent from the loading operation of the program on eBPFlow. There is an option on the loader specific only for operations with maps. Operations with maps via user space do not harm the system's performance because it uses the register interface instead of the processing datapath. Moreover, it can query status information about the processor.

Tools: A set of tools were implemented as part of the eBPFlow infrastructure: an eBPF disassembler, a software emulator, and a CLI application to interact with the eBPF engines. The emulator leverages the uBPF [17]. Software emulator aims to replicate the processor's behavior in software. Furthermore, it enables code testing and debugging with well-known tools such as *gdb*, enabling faster and easier bug detection and correction even before deploying the code.

Communication with host: We chose the NetFPGA's interface nf0 to receive and forward the packets sent from hardware to the host. We do not define the number of cores on the host's CPU to process the packets on userspace. Instead, we leave the operator responsible for defining the number of cores according to the processing demand of the host.

H. Re2c

Re2c [18], [19] is a lexer generator capable of converting regular expressions into fast and optimized finite state machines (FSM). As a result, users can write new protocols in eBPF. We use re2c to convert regular expressions into eBPF-compatible C code. The process encapsulates the FSM generated by Re2c within a function with control pointers. It is also necessary for all states to verify that the pointer's current value is greater than the address at the end of the packet to prevent invalid memory access. After this, the code is packaged in a single file and compiled by the eBPF compiler. Then, a controller sends the generated instructions to eBPFlow, saving them in its instruction memory. From that moment on, the system waits for incoming packets, processes them according to the instructions present in the memory, and performs the appropriate actions according to each packet's content. We do not create an interface between re2c and eBPF-compatible C code, being necessary adaptations for the generated code to work in the eBPFlow. We left the programmer responsible for this task.

V. Network Functions

We have implemented some NFs (Table III) on eBPFlow to demonstrate the offloading of functions and the acceleration in packet processing. For each NF, we present the number of eBPF instructions (#Instructions), the number of C code lines

TABLE III
NETWORK FUNCTIONS IMPLEMENTED ON EBPFLOW

eBPF programs	#Instructions	#LoC in C	State (Yes/No)	# States
Wire	5	7	No	0
Stateful Firewall	23	24	No	0
LPM forwarding	30	26	No	0
DDoS Mitigation	35	57	No	0
BitTorrent Packets	181	86	Yes	5
SQL Injection (Tautology)	143	110	Yes	4
SQL Injection (Sleep)	183	117	Yes	2

(#LoC in C), is there state (Yes/No)?, and the number of states (#States). Here is the description of the NFs:

Wire: acts as a wire connecting adjacent ports in pairs of two. It performs an XOR operation between the input port value and 1, which inverts the least significant bit. This value defines the outgoing packet port. It is the most straightforward application and serves as a performance baseline.

LPM Forwarding (LPMF): forwards packets using the NetFPGA's TCAM module, effectively speeding up longest prefix matching (LPM) operations. In addition, this NF can use up to 32 forwarding rules inserted by the user through the loader.

DDoS Mitigation (DDoS): tries to saturate broadband or overload networking equipment's computational resources, limiting the processing or making unavailable services, servers, and the target network. This NF can analyze random ports of UDP packets. Moreover, it can block the attack on a specific port, dropping the packet and not allowing the attack to have success [20].

Stateful Firewall (SFW): is a network firewall that tracks the status and characteristics of network connections, distinguishing packets for different types of communications and propagating only packets that match the active connections [21].

SQL Injection with Tautology (SQL_TAU): this attack is characterized by the insertion of tautologies in an SQL query, making them manipulable. For example, if the system has the query *SELECT * FROM Users WHERE Id = "username"* where *username* is a user-supplied parameter. If no input filter exists, the attacker can exploit the vulnerability by sending the string *"OR 1 = 1* as a parameter. The resulting query will be *SELECT * FROM Users WHERE Id = "" OR 1 = 1*, which is valid and returns all rows in the *Users* table, since *1 = 1* is always true. It consists of four states that lead to two possible final actions: PASS if there is no attack or DROP if the malicious string is detected before the end of the payload. The first state detects the beginning of the attack, single or double-quotes. The second state detects the presence of spaces and the keyword OR. Finally, the last state detects the end of the attack, 1=1.

SQL Injection with Sleep function (SQL_SLEEP): this attack allows hackers to look for possible SQL vulnerabilities on a server. It uses the User-Agent field of HTTP requests to send an SQL query that calls the function *sleep*, applying a delay in seconds to the current operation. During the delay period, any further requests received run only after the end of the first query, which indicates to the attacker that there are vulnerabilities that allow the insertion of other SQL attacks. The first state detects the presence of the User-Agent keyword or ends processing if it arrives at the end of the payload. The second state looks for the *sleep (* string, which indicates the presence of the attack within the specified field. The processing terminates if a line break occurs before this string. The SQL injection NFs presented above are examples of functions that use regular expressions to efficiently analyze packet payload. This type of analysis is critical today, in which servers store a large amount of valuable data, demanding protection against this and other types of attacks.

BitTorrent Packets (BITP): BitTorrent can cause many simultaneous connections, which can overload the network. This NF detects four BitTorrent packet types based on Strait and Sommer [22]. It is an example of an Application Layer Packet Classifier. The first state is responsible for detecting the patterns' initial character at the beginning of the packet payload. The following four states detect the rest of the strings. This NF only forwards the packet if the patterns are not present at the end of the payload.

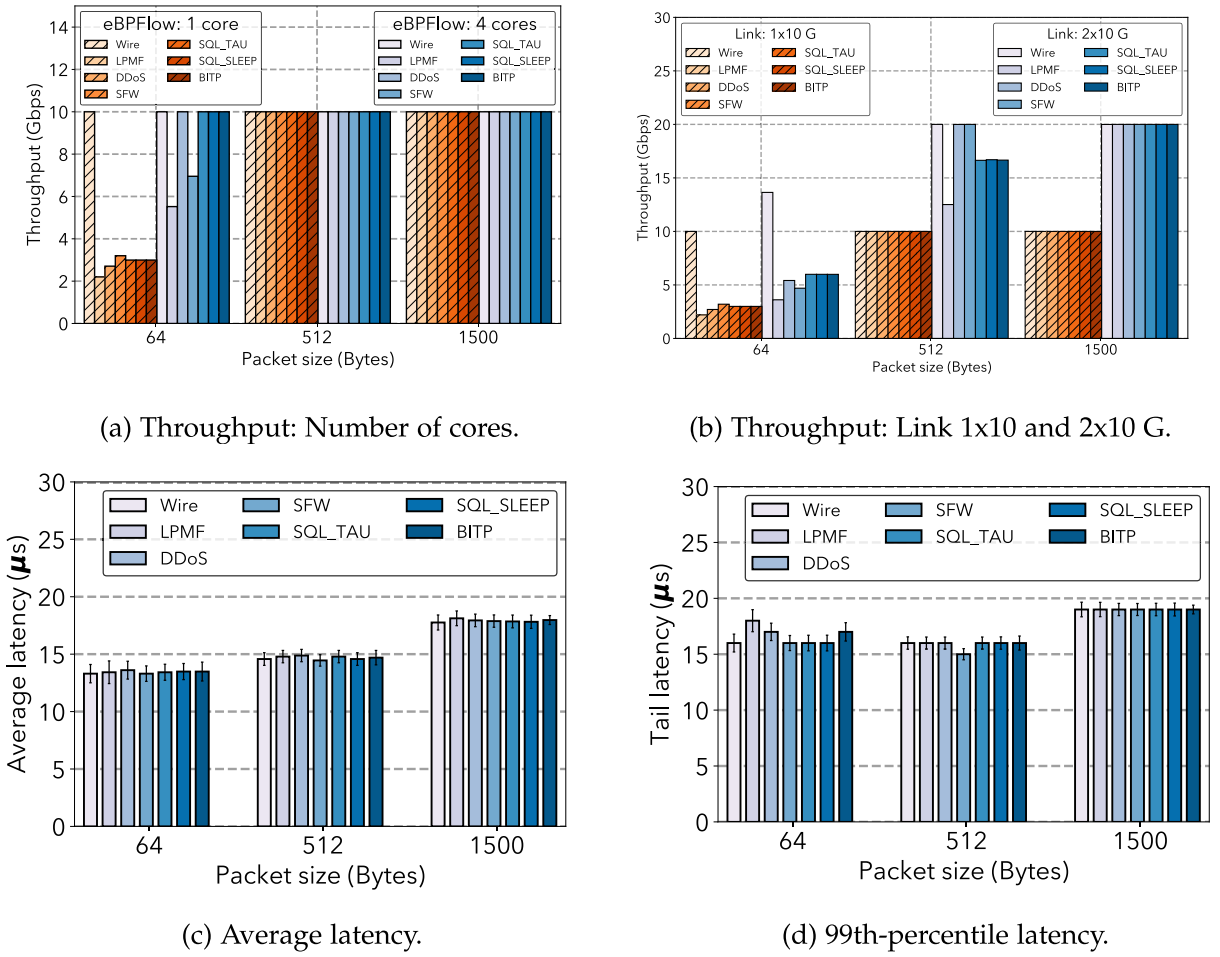
VI. EVALUATION

Here, we present the experimental evaluation of eBPFlow. The testing environment contains one NetFPGA SUME, one server running pktgen-DPDK [23] as a traffic generator, and a custom controller to interact with the eBPFlow's data plane. Our server couples a Netronome Agilio CX SmartNIC and an Intel X710 DA-2 SmartNIC with two 10 Gbps interfaces directly connected to the four NetFPGA SUME ports. We add Intel and Netronome boards on pktgen-DPDK userspace to generate the traffic and to receive the traffic forwarded by NetFPGA SUME, running the eBPFlow design. In addition, the server and machine with coupled NetFPGA SUME have i7-7700 processors clocked at 3.60 GHz containing eight cores and 8 GB of RAM.

A. Throughput

We evaluated the performance of the eBPFlow to packet processing rates 64 bytes (minimum-sized), 512 bytes (middle-sized), 1,500 bytes (maximum-sized), respectively, for network functions described in Section V. Moreover, we used the same parameters to evaluate the system's performance, the number of processing cores per port (one and four), and the number of links generating the traffic (1×10 and 2×10 G).

Figure 6a summarizes the system's throughput according to the number of cores. For packets of 64 bytes using one processing core, all network functions except the Wire achieve throughput less than 4 Gbps, demonstrating that the number of cores affects the system's performance. However, when the



(a) Throughput: Number of cores.

(b) Throughput: Link 1x10 and 2x10 G.

(c) Average latency.

(d) 99th-percentile latency.

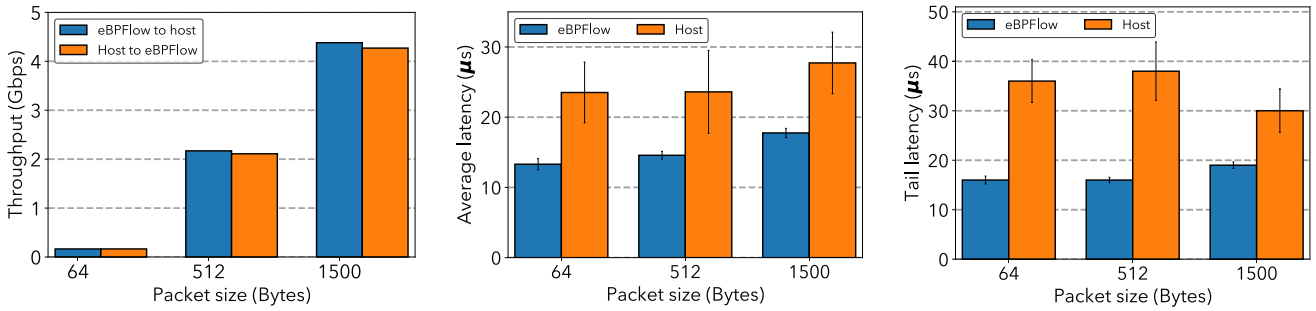
Fig. 6. eBPFlow performance: throughput and latency.

number of cores passes from one to four per port, the throughput of all NFs doubles or achieves line rate (10 Gbps). Wire, DDoS mitigation, SQL Injection attacks, and Bittorrent filter achieve line rate for 64 bytes packets. Moreover, these NFs do not realize operations with maps using CAM/TCAM memories. Stateful Firewall and LPM forwarding have improved throughput with the increase of the number of cores from one to four but yet had throughput reduction due to spent time with operations of access to maps (Section VI-D presents the time in clock cycles and microsecond of each operation).

Figure 6b presents the throughput of the eBPFlow based on the number of links generating traffic per port. We generated traffic using one and two links of 10 Gbps to evaluate the system's performance when stressed. To packets of 64 bytes using one and two links of 10 Gbps, the system had a throughput of less than 6 Gbps per port for all NFs, except the Wire. With two links of 10 Gbps for 512 bytes packets, Wire, DDoS mitigation, and Stateful firewall achieve a throughput of 20 Gbps. On the other hand, SQL Injection attacks and Bittorrent filter achieve a throughput of 16 Gbps, reducing throughput due to the number of instructions because they spend more time processing packets than other NFs and the bottleneck on the FIFOs of the datapath. LPM forwarding has been improved throughput using two links, but the spent time with operations of access to map TCAM harmed the throughput to this network function. Finally, to 1,500 bytes packets, using one and two links, the eBPFlow achieved the maximum throughput (10 and 20 Gbps), respectively, to all NFs without packet loss.

B. Latency

In addition to the throughput, we also measured the average and tail latencies for each NF (Figures 6c and 6d) using *pktgen-DPDK*, with 1 μs precision. pktgen-DPDK measures the end-to-end latency by adding a timestamp on the packet payload. It calculates latency stats, sends the packet to the network, and after the packet returns, gets the timestamp and calculates the time stats. In this experiment, we load the Intel smartNIC on pktgen-DPDK userspace and use one port to send packets and another port to receive the back packets. Moreover, we repeated each experience 33 times for each NF for packet sizes 64, 512, and 1,500 bytes. As expected, latency increased according to the packet size increase because the number of words on the data plane increased, taking more time to run the entire program. All experiments had a latency of less than 20 μs. This metric demonstrates little change in the processing time between same-sized packets for a single



(a) Throughput. (b) Average latency. (c) 99th-percentile latency.

Fig. 7. Communication with host.

NF, leading to reduced jitter. Similarly, the tail latency is close to the average value in almost all cases. The bars in Figures 6c and 6d represent the standard deviation, which was close to zero in all cases.

C. Communication With Host

Throughput: Figures 7a summarizes the throughput of the eBPFlow communicating with the host. We measured the throughput using iPerf3 tool [24] in two directions: eBPFlow communicating with the host (eBPFlow to host) and host communicating with eBPFlow (host to eBPFlow). We executed iPerf's client program on the server and iPerf's server program running on the machine with NetFPGA SUME. Moreover, we generated traffic with TCP packets of 64, 512, and 1,500 bytes in one second. The results show that the system achieves a throughput of less than 5 Gbps to all packet sizes in both directions. It occurs due to context switch and the IP core PCIe v3 with 8-lane of the Xilinx to achieve a maximum transference speed of 5 Gbps [25].

Latency: In this experiment, we measured the average (Figure 7b) and tail (Figure 7c) latencies of the eBPFlow communicating with the host using *pktgen-DPDK*, with 1 μs precision. On a host (machine with coupled NetFPGA), we used Linux's traffic control subsystem called (tc) [26] to receive the packets from board to host and send them back from host to board. Tc is responsible for setup traffic control in the Linux kernel. Moreover, we repeated each experience 33 times for each NF for packet sizes 64, 512, and 1,500 bytes. eBPFlow's latencies (average and tail) had a latency of less than 20 μs with a standard deviation close to zero. While host average and tail latencies were less than 30 and 40 μs with a difference of 10 and 20 μs if compared with eBPFlow's latencies. These results demonstrate that sending the packet from board to host is slower than processing the packet only on the board. However, communication with the host allows dividing the workload between hardware and software and supporting operations not synthesizable.

D. Coprocessor Measurement

This experiment evaluates the coprocessor time spent on each function call (lookup, delete, and update) on eBPFlow. We performed this experiment by adding time registers over the coprocessor's Verilog code. In addition, we created an application to read and add the time values to obtain the time spent after the function call execution. Table IV presents the time in clock cycles (clks) and microseconds (μs) spent on the coprocessor to each eBPF function call. The measurement begins when the coprocessor triggers the call function processing. Each register on code increments its value according to the time spent executing a coprocessor's code-specific functionality. After the coprocessor finishes the function call execution, the application reads and obtains the total time spent on the function call. We compare the times obtained via simulation and tests in the real environment to validate the experiment.

TABLE IV
COPROCESSOR TIME SPENT FUNCTION CALL

	Maps	
Function call	CAM	TCAM
Lookup	13 clks (0.065 μs)	13 clks (0.065 μs)
Delete	15 clks (0.075 μs)	29 clks (0.145 μs)
Update	19 clks (0.095 μs)	33 clks (0.165 μs)

E. eBPFlow Performance on Packet Processing

We evaluated the packet processing capacity of the eBPFlow compared to kernel and Netronome that support offloading of eBPF programs. The kernel runs eBPF programs in software, while Netronome uses a SmartNIC. To compare both systems, we choose the network functions SQL Sleep because it is the eBPF program with more significant numbers of instructions (Table III). We executed this NF on the three systems. Moreover, we measured the throughput in millions of packets per second (Mpps) to evaluate the packet processing power between software (using kernel Linux 5.0.4) and hardware (using Netronome and eBPFlow). We generated packet rates of 64, 512, and 1500 bytes using pktgen-DPDK.

On the kernel, we insert a map structure in each network function to count the number of processed packets and measure this information using the xdp-stats tool available in [27] for each experiment. We evaluated Netronome SmartNIC with their stat_watch.py tool [28]. Figure 8 presents the throughput in Mbps of each system, respectively, to evaluate

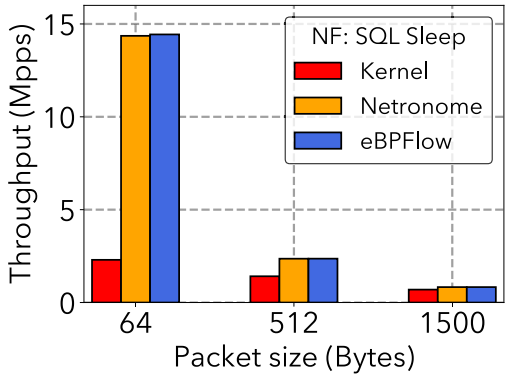


Fig. 8. Systems performance on packet processing.

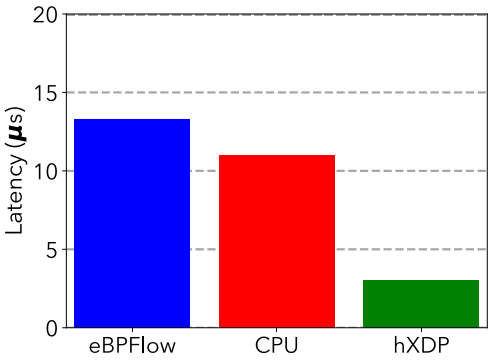


Fig. 10. Latency: eBPFlow, CPU, and hXDP.

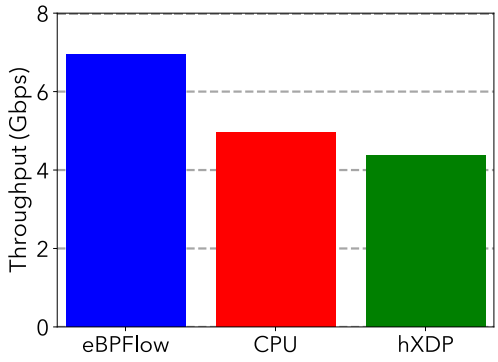


Fig. 9. Throughput: eBPFlow, CPU, and hXDP.

network function. Netronome and eBPFlow processes approximately 12.05, 0.87, 0.15 Mpps more than the kernel to packet sizes 64, 512, and 1500 bytes. The packet processing between Netronome and eBPFlow is similar to all packet sizes. However, eBPFlow provides functionalities not available on Netronome, such as parallelism per port using eBPF cores reserved per port and parallelism on packet forwarding using an integrated output crossbar on the system's data plane. In the Related Work Section (Section VII), we describe more details between both systems.

F. eBPFlow Performance Compared to Other Systems

We compared the performance of two systems (hXDP and CPU x86@3.7 GHz) that are similar to eBPFlow. hXDP runs eBPF code on the NetFPGA. CPU means we execute code in the kernel. Kernel executes eBPF code on the CPU. We evaluated the throughput and latency of the systems to 64 bytes packets (minimum-size) using a firewall as a network function. We choose the firewall as NF based on experiments of [11]. Moreover, to compare the systems fairly, for eBPFlow, we used one physical port to receive packets, one physical port to send packets, and four eBPF engines. This is the same setup applied to CPU and hXDP [11].

Throughput: Figure 9 presents a throughput comparison of the systems. hXDP achieves a throughput of 4.36 Gbps. CPU x86@3.7 GHz gets a throughput of 4.97 Gbps. Finally, eBPFlow obtains a throughput of 6.95 Gbps. This result demonstrates that the eBPFlow has a processing performance improvement of 2.59 and 1.98 Gbps over hXDP and CPU x86@3.7 GHz, respectively.

Latency: Figure 10 presents a latency comparison of the systems on packet forwarding. CPU x86@3.7 GHz gets a latency of 11 μs. hXDP achieves a latency of 3 μs. Finally, eBPFlow obtains a latency of 13.30 μs. This result demonstrates that the eBPFlow spends more time on packet forwarding 2.3 μs and 10.3 μs about CPU x86@3.7 GHz and hXDP, respectively. eBPFlow overcomes this limitation by providing parallelism in design and processing more packets than both systems.

G. Power

When idle, the NetFPGA consumes 16 W. However, when we synthesize eBPFlow, the power consumption is 22 W regardless of the packet rate or running program [21]. Devices as Netronome [29], [30], Intel Core i7-7700 [31], 1U rack-mount x86 [32], and P4 Wedge 100BF-32X [33] have power consumption of approximately 25-40 W, 65 W, 300-350 W, and 436 W, which demonstrates that eBPFlow saves power in comparison to the listed devices.

VII. Related Work

EBPFlow is the first SW/HW system that seamlessly offloads NFs and can process packet headers and payloads to the best of our knowledge. eBPFlow enables the programming of stateful and stateless NFs, and can dynamically modify the parser, matching, and actions at runtime.

Programmable networks. The OpenFlow [34] standard, although being the most adopted SDN architecture [35], has limitations. Its matching structure cannot do inequality, complement (not operation), or range matching. On the other hand, eBPFlow allows for logical expressions (not, and, or) and range comparison (>,<). Liu et al. [36] developed the CLARA, a network slicing architecture that uses NFV concepts and reinforcement learning algorithms for resource allocation management. Finally, Yen et al. [37] proposed the Lemur, a system that places and executes NF chains across heterogeneous hardware while meeting service-level objectives

(SLOs) in NFV. It receives as input a high-level description of multiple NF chain DAGs and their associated SLOs. As output, the system returns a placement configuration for each NF chain and coordination code, ensuring that the NF executes on the appropriate hardware element specified by the placement.

High level domain-specific languages. The P4 programming language P4 [38] adopts the match-action abstraction model. Therefore, it is possible to use the P4 language to generate eBPF instructions using the compiler from P4 to eBPF [39]. Domino [40] is a high-level language compiled into Banzai, a low-level machine model designed for line-rate switches. Although P4 and Domino include small and fast registers to store states, they provide a restricted functionality for many stateful functions. Kfoury et al. [41] present an exhaustive survey about P4 programmable data plane switches highlighting subjects like taxonomy, applications, challenges, and trends.

BPF related. BPFabric [42] proposed a software platform that allows protocol-independent packet processing. It uses eBPF instructions to define the packet processing and forwarding in the data plane. BPFabric was initially implemented over a Linux raw socket interface and later adapted over the DPDK. hXDP [11] is a system to run Linux's XDP programs on an FPGA. hXDP is similar to the eBPFlow, and it executes XDP code. The hXDP design changed the load/store instructions and introduced three-operand instructions. The hXDP design does not support network functions offloading at runtime like the eBPFlow, which has many processing cores with an instruction memory containing a double buffer system. FFShark [43] is an implementation of the Wireshark in an FPGA. It contains eBPF cores to execute written filters in the PCAP filtering language. However, FFShark does not provide instructions parallelism with eBPF cores containing a 5-stages pipeline coupled on the cores. Katran [44] is an open-source eBPF load-balancer application provided by Facebook, showing eBPF's adoption trend in the industry. Chaining-Box [45] is a Service Function Chaining (SFC) architecture where all the SFC functionality are implemented, in a fully transparent manner, as a sequence of eBPF stages.

FPGA related. P4FPGA [46] is a platform developed in hardware that performs the conversion of P4 programs to Verilog. P4-To-VHDL [47] is a tool that converts a P4 description to a synthesizable VHDL code suitable for the FPGA implementation. ClickNP [48] also focuses on increasing programmability flexibility. It provides a declarative language called ClickNP. ClickNP can be compiled into an intermediate hardware description language (HDL) and synthesized on the FPGA. Zang et al. [49] proposed a distributed-agent NFV system that supports Service Function Chaining (SFC) of FPGAs and microprocessors. The system works with an agent helping the partial reconfiguration core to control the dynamic reconfiguration of middlebox functions on FPGAs. FlowBaze [21] is an FPGA-based SmartNIC that allows stateful packet processing in hardware by programming using Extended Finite State Machines (EFSM). However, it cannot operate on the packet payload and only supports storing 64 states. eBPFlow does not have these limitations because the states and transitions of an FSM are transformed into instructions. PANIC [50] is an FPGA based on Reconfigure Match Action (RMT) switches that schedule the order in which the packets are processed and distribute the packets across the different compute units. Eran et al. [51] proposed the NICA, an FPGA-based NIC server acceleration system that supports software abstractions via functional units for application acceleration in cloud systems. NICA was implemented on Mellanox and integrated with an abstraction denominated ikernel (inline kernel), which represents an Acceleration Functional Unit (AFU) in a user program. Finally, other related works that use FPGAs [52], [53], [54].

Smart NICs. Netronome [8] provides a SmartNIC programmed with eBPF instructions. Some features and commands are specific to the kernel and firmware version, generating incompatibility on the network. When an update is released, firmware or kernel needs to be updated manually, generating failures and hindering the management of the network. Nonetheless, eBPFlow does not have these dependencies. It is independent and seamless with other technologies, e.g., the Linux kernel. Furthermore, to load a program into Netronome NIC, the code has to pass a verifier which does not allow back-edge jump (e.g., for, while), so the SmartNIC can not compute DPI NFs with different packet sizes. Netronome NICs have very low port density, with at most two ports per NIC. On the other hand, the eBPFlow prototype has 12 physical ports. Moreover, eBPFlow design includes specific memory hardware, such as CAM and TCAM, to handle stateful NFs. Dyssect [55] disaggregates the states of NFs and allows the offloading of stateful NFs to programmable NICs.

VIII. Conclusion

eBPFlow is a packet processing platform targeted for high-performance data planes composed of hardware and software components built on top of the NetFPGA SUME platform. The system dynamically supports parsing, matching, and actions operations through eBPF instructions. Moreover, eBPFlow is protocol-independent, and it allows the use of new fields, easing the adoption of new protocols and services. In addition, eBPFlow can process both the packet header and payload at the line rate. Another functionality fundamental of the system is to support programmable network functions at runtime, modifying the flow processing logic by swapping the image of the eBPF program using an instruction memory with a double buffer system. We designed and implemented multiple eBPF virtual machines in hardware at its core to support these functionalities, communicating with userspace tools. The eBPFlow's repository is publicly available on Github [56].

References

[1] L. Linguaglossa et al., "Survey of performance acceleration techniques for network function virtualization," *Proc. IEEE*, vol. 107, no. 4, pp. 746–764, Apr. 2019.

[2] E. Kaljic, A. Maric, P. Njemcevic, and M. Hadzialic, "A survey on data plane flexibility and programmability in software-defined networking," *IEEE Access*, vol. 7, pp. 47804–47840, 2019.

[3] R. Bifulco and G. Rétvári, "A survey on the programmable data plane: Abstractions, architectures, and open problems," in *Proc. IEEE 19th Int. Conf. High Perform. Switching Routing (HPSR)*, Jun. 2018, pp. 1–7.

[4] L. B. D. Silva et al., "READY: A fine-grained multithreading overlay framework for modern CPU-FPGA dataflow applications," *ACM Trans. Embedded Comput. Syst.*, vol. 18, no. 5, pp. 1–20, Oct. 2019, doi: 10.1145/3358187.

[5] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications," *ACM Comput. Surv.*, vol. 53, no. 1, pp. 1–36, Feb. 2020, doi: 10.1145/3371038.

[6] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, Sep. 2014.

[7] R. D. G. Pacífico, M. S. Castanho, L. F. M. Vieira, M. A. M. Vieira, L. F. S. Duarte, and J. A. M. Nacif, "Application layer packet classifier in hardware," in *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manage. (IM)*, May 2021, pp. 515–522.

[8] D. Beckett, J. Joubert, and S. Horman, "Host dataplane acceleration (HDA)," in *Proc. ACM Special Interest Group Data Commun. Tutorial (SIGCOMM)*. Budapest, Hungary: ACM, 2018.

[9] J. Schulist, D. Borkmann, and A. Starovoitov. (1993). *Linux Socket Filtering aka Berkeley Packet Filter (BPF)*. Accessed: Dec. 15, 2017. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/filter.txt

[10] A. W. Moore. (2007). *NetFPGA Project*. Accessed: Dec. 21, 2017. [Online]. Available: https://www.netfpga.org

[11] M. S. Brunella et al., "hXDP: Efficient software packet processing on FPGA NICs," in *Proc. 14th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2020, pp. 973–990.

[12] J. W. Lockwood et al., "NetFPGA—An open platform for gigabit-rate network switching and routing," in *Proc. IEEE Int. Conf. Microelectron. Syst. Educ. (MSE)*. Washington, DC, USA: IEEE Computer Society, Jun. 2007, pp. 160–161, doi: 10.1109/MSE.2007.69.

[13] Xilinx. (2010). *Virtex-7 Family Overview*. [Online]. Available: https://www.xilinx.com

[14] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann, 2011.

[15] K. Locke, "Parameterizable content-addressable memory," Xilinx, San Jose, CA, USA, Appl. Note XAPP1151, 2011.

[16] Xilinx. (Jan. 2018). *Xilinx Core Generator System*. [Online]. Available: https://www.xilinx.com/products/design-tools/cor egen.html

[17] Big Switch Networks. (2020). *Userspace eBPF VM*. Accessed: Jun. 22, 2020. [Online]. Available: https://github.com/iovisor/ubpf

[18] RE2C Organization. (1993). *RE2C*. Accessed: Jun. 18, 2020. [Online]. Available: https://re2c.org/

[19] P. Bumbulis and D. D. Cowan, "RE2C: A more versatile scanner generator," *ACM Lett. Program. Lang. Syst.*, vol. 2, nos. 1–4, pp. 70–84, Mar. 1993, doi: 10.1145/176454.176487.

[20] G. Bertin, "XDP in practice: Integrating XDP into our DDoS mitigation," in *Proc. Tech. Conf. Linux Netw.*, 2017, p. 5. [Online]. Available: https://www.netdevconf.org/2.1/papers/Gilberto_Bertin_XDP_in_practice.pdf

[21] S. Pontarelli et al., "FlowBlaze: Stateful packet processing in hardware," in *Proc. 16th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. Boston, MA, USA: USENIX Association, 2019, pp. 531–548.

[22] M. Strait and E. Sommer. (2003). *L7 Filter—Bittorrent*. Accessed: Jun. 22, 2020. [Online]. Available: http://l7-filter.sourceforge.net/layer7-protocols/protocols/bittorrent.pat

[23] D. Turull, P. Sjödin, and R. Olsson, "Pktgen: Measuring performance on high speed networks," *Comput. Commun.*, vol. 82, pp. 39–48, May 2016.

[24] iPerf 3. (2014). *Documentation iPerf 3*. Accessed: Dec. 31, 2021. [Online]. Available: https://iperf.fr/

[25] Xilinx. (2014). *7 Series FPGAS Integrated Block for PCI Express V3.0*. Accessed: Dec. 31, 2021. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/pcie_7x/v3_0/pg054-7seriespcie.pdf

[26] Linux. (2021). *TC(8) Linux Manual Page*. Accessed: Jan. 3, 2021. [Online]. Available: https://man7.org/linux/man-pages/man8/tc.8.html

[27] XDP. (2022). *XDP Project: XDP Hands-on Tutorial*. Accessed: Feb. 9, 2022. [Online]. Available: https://github.com/xdp-project/xdp-tutorial

[28] Netronome. (2022). *Netronome Flow Processor (NFP) Kernel Drivers*. Accessed: Feb. 9, 2022. [Online]. Available: https://github.com/Netronome/nfp-drv-kmods/blob/master/tools/stat_watch.py

[29] Open-NFP. (2017). *SmartNIC Programming Models*. Accessed: Dec. 27, 2021. [Online]. Available: https://open-nfp.org/static/pdfs/dxdd-eu-smartnic-prog-models-2017-06-07.pdf

[30] J. Hypolite, J. Sonchack, S. Hershkop, N. Dautenhahn, A. DeHon, and J. M. Smith, "DeepMatch: Practical deep packet inspection in the data plane using network processors," in *Proc. 16th Int. Conf. Emerg. Netw. Exp. Technol.*, Nov. 2020, pp. 336–350.

[31] CPU Agent. (2016). *Intel Core i7-7700 Review*. Accessed: Dec. 27, 2021. [Online]. Available: https://www.cpuagent.com/cpu/intel-core-i7-7700/summary/nvidia-geforce-gtx-980-ti

[32] Vertatique. (2015). *Average Power Use Per Server*. Accessed: Dec. 27, 2021. [Online]. Available: https://www.vertatique.com/average-power-use-server

[33] Edge Core. (2017). *Wedge 100–32x Datasheet*. Accessed: Dec. 27, 2021. [Online]. Available: https://www.edge-core.com/_upload/images/Wedge_100-32X_DS_R04_20170615.pdf

[34] N. McKeown et al., "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008, doi: 10.1145/1355734.1355746.

[35] H. Moura, A. R. Alves, J. R. A. Borges, D. F. Macedo, and M. A. M. Vieira, "Ethanol: A software-defined wireless networking architecture for IEEE 802.11 networks," *Comput. Commun.*, vol. 149, pp. 176–188, Jan. 2020.

[36] Y. Liu, J. Ding, Z.-L. Zhang, and X. Liu, "CLARA: A constrained reinforcement learning based resource allocation framework for network slicing," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2021, pp. 1427–1437.

[37] J. Yen, J. Wang, S. Supittayapornpong, M. A. M. Vieira, R. Govindan, and B. Raghavan, "Meeting SLOs in cross-platform NFV," in *Proc. 16th Int. Conf. Emerg. Netw. Exp. Technol.* New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 509–523, doi: 10.1145/3386367.3431292.

[38] P. Bosshart et al., "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014, doi: 10.1145/2656877.2656890.

[39] M. Budiu. (2015). *Compiling P4 to eBPF*. [Online]. Available: https://github.com/iovisor/bcc/tree/master/src/cc/frontends/p4

[40] A. Sivaraman et al., "Packet transactions: High-level programming for line-rate switches," in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, Aug. 2016, pp. 15–28, doi: 10.1145/2934872.2934900.

[41] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An exhaustive survey on P4 programmable data plane switches: Taxonomy, applications, challenges, and future trends," *IEEE Access*, vol. 9, pp. 87094–87155, 2021.

[42] S. Jouet and D. P. Pezaros, "BPFabric: Data plane programmability for software defined networks," in *Proc. ACM/IEEE Symp. Architectures Netw. Commun. Syst. (ANCS)*. Piscataway, NJ, USA: IEEE Press, May 2017, pp. 38–48, doi: 10.1109/ANCS.2017.14.

[43] J. C. Vega, M. A. Merlini, and P. Chow, "FFShark: A 100G FPGA implementation of BPF filtering for wireshark," in *Proc. IEEE 28th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2020, pp. 47–55.

[44] Facebook. (2018). *Katran Source Code Repository*. [Online]. Available: https://github.com/facebookincubator/katran

[45] M. S. Castanho, C. K. Dominicini, M. Martinello, and M. A. M. Vieira, "Chaining-box: A transparent service function chaining architecture leveraging BPF," *IEEE Trans. Netw. Service Manage.*, vol. 19, no. 1, pp. 497–509, Mar. 2022.

[46] H. Wang et al., "P4FPGA: A rapid prototyping framework for P4," in *Proc. Symp. SDN Res.*, New York, NY, USA, Apr. 2017, pp. 122–135, doi: 10.1145/3050220.3050234.

[47] P. Benáček, V. Puš, H. Kubátová, and T. Čejka, "P4-To-VHDL: Automatic generation of high-speed input and output network blocks," *Microprocessors Microsyst.*, vol. 56, pp. 22–33, Feb. 2018.

[48] B. Li et al., "ClickNP: Highly flexible and high performance network processing with reconfigurable hardware," in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, Aug. 2016, pp. 1–14, doi: 10.1145/2934872.2934897.

[49] X. Zhang and R. Tessier, "Service chaining for heterogeneous middleboxes," in *Proc. Int. Conf. Field-Program. Technol. (ICFPT)*, Dec. 2020, pp. 263–267.

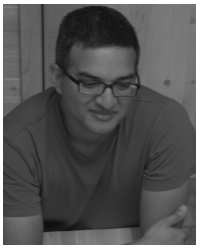
[50] J. Lin, K. Patel, B. E. Stephens, A. Sivaraman, and A. Akella, "PANIC: A high-performance programmable NIC for multi-tenant networks," in *Proc. 14th USENIX Conf. Oper. Syst. Design Implement.*, 2020, pp. 243–259.

[51] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein, "NICA: An infrastructure for inline acceleration of network applications," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 345–362.
[52] R. D. Pacífico et al., "Bloomtime: Space-efficient stateful tracking of time-dependent network performance metrics," *Telecommun. Syst.*, vol. 74, pp. 201–223, Feb. 2020.
[53] R. D. G. Pacífico, M. A. M. Vieira, L. F. S. Duarte, and J. A. M. Nacif, "Function as a service offloaded to a SmartNIC," in *Proc. IEEE Latin-Amer. Conf. Commun. (LATINCOM)*, Nov. 2022, pp. 1–6.
[54] R. Pacífico, P. Goulart, A. B. Vieira, M. A. M. Vieira, and J. A. M. Nacif, "Hardware modules for packet interarrival time monitoring for software defined measurements," in *Proc. IEEE 41st Conf. Local Comput. Netw. (LCN)*, Nov. 2016, pp. 188–191.
[55] F. B. Carvalho, R. A. Ferreira, Ì. Cunha, M. A. M. Vieira, and M. K. Ramanathan, "State disaggregation for dynamic scaling of network functions," *IEEE/ACM Trans. Netw.*, early access, Jun. 12, 2023, doi: 10.1109/TNET.2023.3282562.
[56] eBPFlow. (2021). *Repository of the eBPFlow*. Accessed: Feb. 14, 2022. [Online]. Available: https://github.com/racyusdelanoo/ebpflow

Racyus D. G. Pacífico received the B.S. degree from IF SUDESTE/MG in 2014 and the M.S. degree in computer science from Universidade Federal de Viçosa (UFV) in 2016. He is currently pursuing the Ph.D. degree in computer science with Universidade Federal de Minas Gerais (UFMG), Brazil. His research interests include reconfigurable computing, computer architecture, and computer networking.

Lucas F. S. Duarte received the B.S. degree in computer science from Universidade Federal de Viçosa (UFV) in 2021. He has experience in computer networking and developing mobile solutions for financial companies.

Luiz F. M. Vieira received the bachelor's and M.S. degrees from Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, and the Ph.D. degree in computer science from the University of California Los Angeles (UCLA). He is currently an Associate Professor with the Computer Science Department (DCC), UFMG. He has received several awards, including Fulbright Scholarship, CAPES Best Dissertation, and SBRC Best Thesis and Dissertation. He is also awarded with a productivity in research scholarship from CNPq Research Agency. He has served on the technical committee of networking conferences, such as IEEE INFOCOM. He is also an Editor of *Ad Hoc Networks* (Elsevier). His research interests include computer networking and wireless networks.



Barath Raghavan received the B.S. degree in electrical engineering and computer science from UC Berkeley in 2002 and the Ph.D. degree in computer science from UC San Diego in 2009. Before joining University of Southern California (USC) in 2018, he split his time between industry and academia working on a wide range of projects in core computer science areas, such as computer networking, security, and distributed systems, and on socially-focused, topics such as rural internet access and sustainable agriculture.

José A. M. Nacif (Member, IEEE) received the Ph.D. degree in computer science from Universidade Federal de Minas Gerais. He has been a Professor in computer science with the Universidade Federal de Viçosa, Brazil, since 2010, where he is currently the Leader of the Computer System Engineering Laboratory and a CNPq Productivity Fellow. His research interests include the Internet of Things, reconfigurable computing, and electronic design automation.

Marcos A. M. Vieira received the bachelor's and M.S. degrees from Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, and the M.S. and Ph.D. degrees in computer science from the University of Southern California (USC). He has been an Assistant Professor in computer science with UFMG since 2011. He was a Visiting Professor with USC in 2018 and 2019. He is a CNPq (Brazilian Research Agency similar to NSF) Research Fellowship—level 1D. His research interests include computer networks, wireless networks, and cyber-physical systems, ranging from sensor networks to the Internet of Things.