# VeBPF Many-Core Architecture for Network Functions in FPGA-based SmartNICs and IoT

Zaid Tahir[a] Ahmed Sanaullah[b] Sahan Bandara[a] Ulrich Drepper[b] Martin Herbordt[a]
[a]CAAD Lab, Electrical and Computer Engineering, Boston University, USA. - [b]Red Hat Inc.
zaidt@bu.edu asanaull@redhat.com sahanb@bu.edu drepper@redhat.com herbordt@bu.edu

***Abstract*—FPGA-based SmartNICs and IoT devices integrated with soft-processors for executing network functions have been introduced to overcome hardware-reconfigurability limitations in DPUs (Data Processing Units) and MCUs (Microcontroller Units), respectively. However, existing FPGA-based SmartNICs and IoT devices lack a highly configurable many-core architecture that specializes in network packet processing.**

**This work introduces a resource-optimized highly configurable VeBPF (Verilog eBPF) many-core architecture built upon VeBPF CPU cores that we have developed for specialized network packet processing in FPGAs. These VeBPF cores are eBPF ISA compliant and have been developed in Verilog HDL for easy integration with existing FPGA IP blocks/subsystems. The VeBPF many-core architecture executes multiple eBPF rules on multiple VeBPF cores in-parallel for low-latency network packet processing. Due to the highly configurable hardware design of this VeBPF many-core architecture, any number of VeBPF cores can be instantiated by assigning a parameter $N_{VeBPF}$ in the Verilog code of the VeBPF many-core architecture and any number of eBPF rules can be uploaded, with FPGA resources as the only constraint. The proposed VeBPF many-core architecture has been designed to process eBPF rules faster if $N_{VeBPF}$ is increased and the eBPF rules can be dynamically changed during run-time without requiring new bitstreams. It uses various hardware and computer architecture optimizations to support its implementation on low-end FPGAs-based IoT devices along with high-end FPGA-based SmartNICs, for network packet processing. We have also developed automatic-testing and simulation frameworks for the proposed VeBPF many-core architecture, using the latest open-source tools like Python and Cocotb. We have released the Verilog HDL code for VeBPF core development, VeBPF many-core architecture, C software libraries for RISC-V control of m-plane (management-plane) of the VeBPF many-core architecture and the simulators as an open-source contribution for further advancement of FPGAs in many-core architectures, eBPF, SmartNICs, IoT, cybersecurity and communication.**

***Index Terms*—FPGA, Many-core, Multi-core, eBPF, SmartNIC, IoT, RISC-V, Network Communication, Cybersecurity.**

## I. Introduction and Background

As is well-known, advances in process technology have run up against limitations in Dennard scaling and Moore's Law resulting in fundamental changes to CPU architecture, the most obvious being the emergence of multicore. Other fundamental shifts in computing, such as to data-centers/clouds and edge-based IoT devices have exposed new limitations in CPU architectures, especially when cost and power are considered. Money, energy and time is lost with every cycle of cloud host CPUs spent on network communication and other tasks that are unrelated to the user applications.

In order to decouple host CPUs from the computational loads of network packet processing, SmartNICs (Smart Network Interface Cards) [1] have been introduced that perform network functions. Since SmartNICs need high computational capability, throughput, and energy efficiency, many-core processors have also been integrated into the latest SmartNICs such as DPUs (Data Processing Units) [2] or FPGA-based SmartNICs [3].

DPU-based SmartNICs suffer from the limitation that the hardware is fixed and if new hardware features are required, e.g., due to upgrades in protocols and interfaces, the existing DPU may need to be replaced, or, at least, lose relative performance. FPGA-based SmartNICs have long filled a niche in this space as they provide high-throughput communication while having reconfigurable hardware [4]–[7]. Often this configurability is used, at least partially, to implement dedicated soft processors [3], [8], [9]. The issue addressed here is the design of these processors: many-core architectures available for FPGAs are mostly based on homogeneous general purpose processors such as RISC-V [10], [8], or, if they are heterogeneous many-core architectures [11], they still involve various general purpose processors not specialized for network packet processing.

For network packet processing eBPF ISA (Instruction Set Architecture) [12] provides specialized instructions for network functions, which is one of the reasons eBPF compilers and tool-chains are native to UNIX-like operating systems with eBPF bytecode executed in the kernel space. Many technology companies have included eBPF into their software stacks [13]. Due to these advantages eBPF ISA compliant soft-processors have been developed like [14], [15]. Both, however are single processor solutions. [14] uses a custom compiler to convert native eBPF instructions to special VLIW instructions; this introduces issues of maintaining correct versions of compilers and drivers [16]–[18]. [15] is written in Migen [19], which may be difficult to integrate with IP blocks written in HDLs.

In the existing FPGA-based SmartNICs such as [1], [3], [20], [21], the network packet processing frameworks are aimed at high-end FPGAs for cloud-based deployments. This leaves a void for network packet processing frameworks for low-end FPGAs essential to IoT devices [22], [23]. These FPGA-based IoT devices have easy access to fast wireless networks due to the maturity of 5G infrastructure. The communication loads on these FPGA-based IoT devices have increased tremendously, especially with AI integrated in many

applications. Such high connectivity also introduces threats of malicious cyberattacks [24]. Hence, low-end FPGAs used as IoT devices also need a network packet processing framework that takes the network packet processing load off the main soft-processor. The problem is that these low-end FPGA-based IoT devices lack such a resource-optimized network packet processing framework.

Due to the lack of many-core architectures specialized in network packet processing and in order to offload network communication processing loads for both high-end FPGA-based SmartNICs and low-end FPGA-based IoT devices, this paper presents a highly configurable and resource-optimized VeBPF many-core architecture. The PE (Processing Element) of the proposed many-core architecture, the VeBPF CPU core, has been developed to specialize in network packet processing.

The contributions of this work are summarized as follows:

- The PE of the proposed many-core architecture, the VeBPF core, has been developed to be eBPF ISA [12] compliant, making it specialized for network packet processing. It is implemented in Verilog HDL, ensuring portability and easy integration with existing IP blocks.
- We have developed various design optimizations for low-latency network packet processing, e.g., single clock cycle reprogramming of the VeBPF core, making it possible to switch between multiple eBPF rules by changing the VeBPF core PC externally in just a single clock cycle.
- Due to the flexible and optimized hardware design of the VeBPF many-core architecture, any number of VeBPF cores can be instantiated and any number of eBPF rules can be uploaded.
- The scheduling and arbitration logic in the proposed VeBPF many-core architecture has been designed in such a way that increasing the number of VeBPF cores leads to faster processing of the eBPF rules on the network packets.
- We have designed the shared data and control buses of the VeBPF many-core designs to minimize resource usage. Resource-intensive communication modules such as NOCs and reconfigurable match action pipelines (RMTs) were intentionally avoided.
- Unlike existing network packet processing frameworks in FPGA-based SmartNICs [1], [3], where the full network packets are held in temporary registers till processing on them is completed, the VeBPF many-core architecture has been designed such that the full network packets are written to memory as soon as they arrive.
- We also developed an automatic testing framework for the developing and upgrading the VeBPF CPU core, and a complete simulation framework for the VeBPF many-core designs.

We have released the Verilog HDL code for VeBPF core[1] development and the VeBPF many-core design.[2] This includes the C libraries for RISC-V control of the management-plane, as well as the simulation and testing frameworks.

In the coming sections we elaborate on the computer architecture design details of the VeBPF core development and the overall many-core hardware architecture. We also present a firewall application implemented using VeBPF on a FPGA-based IoT device. We compare resource usage of the VeBPF cores with similar PEs.

## II. VeBPF Many-Core Architecture

This section presents a detailed overview of the hardware architecture of the proposed VeBPF many-core architecture and its building blocks, starting from the many-core PE (the VeBPF CPU core) and rest of the modules that make up this VeBPF many-core architecture.

### A. VeBPF CPU Core

One of the major goals of the proposed VeBPF many-core architecture is to ensure that the PE, the basic compute unit of the many-core architecture, is specialized for network packet processing and the VeBPF many-core architecture is readily usable without requiring custom drivers and compilers. It is due to these reasons that we developed the VeBPF CPU core, which is the PE for our VeBPF many-core architecture, to be eBPF ISA [12] compliant since eBPF is native to UNIX-like OS and is widely used by technology companies in their software stacks [13].

We deliberately chose Verilog as the HDL language for the development of the VeBPF CPU core, after doing research and development on other higher-level HDL languages like Migen [19], [15], we found that these higher-level HDL languages aren't readily portable with other lower-level HDL languages like Verilog, VHDL and System Verilog. Hence developing the VeBPF CPU core in Verilog HDL would make our VeBPF many-core architecture quite portable.

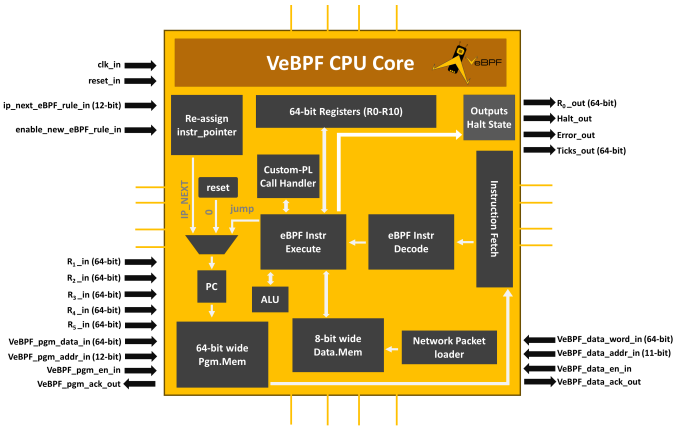


Fig. 1: VeBPF CPU core computer architecture overview.

*1) VeBPF CPU core computer architecture:* The VeBPF CPU core is depicted in Fig. 1, has a Harvard architecture as seen from the separate program and data memory blocks. The data memory is 8-bits wide and its depth is adjustable.

[1] https://github.com/zaidtahirbutt/VeBPF
[2] https://github.com/zaidtahirbutt/VebpfManyCore

We normally select the depth of the data memory to be equal to that of the of the maximum length of the network packet headers instead of setting it to full network packet length because the VeBPF core only needs to process the headers of the network packets as is usually done in eBPF programs. The program memory is 64-bits wide and its depth is also adjustable. The VeBPF core has 11 64-bit registers (*R0 – R10*). The registers *R1 – R5* are used as input registers and they are not cleared upon *reset* given to the VeBPF core. The register *R0* is used as the output register.

As soon as the *reset_in* signal is set *LOW*, the VeBPF core starts executing the eBPF instructions that are read from the program memory based on the value of *PC* and sent to the *Instruction Fetch* module, which then get decoded and executed by their respective blocks as shown in Fig. 1. As soon as the eBPF *exit* instruction is processed and the eBPF program is finished, the *Halt_out* output port becomes *HIGH* with the output result available at *R0* register. If any error had occurred during processing, the *Error_out* signal becomes *HIGH*. The total clock ticks taken to run the eBPF program are also available at *Ticks_out* output port. The eBPF *call* instruction in Linux Kernel is used to call *Helper_Functions* while in case of the VeBPF core, the *call* eBPF instruction is directed to the *Custom-PL Call Handler* block where users can implement application specific custom hardware accelerators.

*2) Single clock cycle eBPF rule switching:* We have developed an important functional optimization to the computer architecture of the VeBPF core where the instruction pointer that reads the eBPF program instruction from the program memory, is settable from outside the VeBPF core while the VeBPF core is in reset state, for the purpose of reprogramming the VeBPF CPU core to a different eBPF rule within a single clock cycle. The block labelled as *Re-assign instr_pointer* depicted in Fig. 1 is in charge of doing that. The benefit of this optimization is that the VeBPF core is reprogrammable to a different eBPF rule with just a change of its instruction pointer to the value where the particular eBPF rule is located in the program memory of the VeBPF core in a single clock cycle. This optimization saves tremendous amount of clock cycles versus what it would have taken if, for each eBPF rule, that rule would have had to be uploaded in the program memory of the VeBPF core first before its execution. It is pertinent to mention here that the VeBPF many-core multi-rule program loader module Fig. 5 uploads all the eBPF rules to the program memories of all VeBPF cores so that all the VeBPF cores are reprogrammable to any eBPF rule with just a change in their instruction pointers in a single clock cycle by the VeBPF many-core multi-rule scheduler module Fig. 6.

*3) VeBPF CPU core resource usage:* Table-I shows the FPGA resource utilization of a single VeBPF core compared with PE of a RISC-V-based many-core architecture [8] and a eBPF ISA compliant core that needs custom compilers and drivers [14]. Our VeBPF core requires significantly less FPGA resources as seen in Table-I which falls in-line with the goal of optimized computer architecture design of the PE of our VeBPF many-core architecture so that we are able to target both low-end FPGAs for IoT deployments and high-end FPGAs for SmartNIC deployments.

TABLE I: FPGA Resource Utilization for Various PE Cores

| Single PE Core | LUTs | FFs | BRAMs |
|---|---|---|---|
| VeBPF core | 3500 | 1600 | 1.5 |
| RISC-V PE [8] | 7878 | 1944 | 20 |
| SEPHIROT [14] | 27000 | 4000 | - |

### *B. VeBPF Many-Core Architecture Overview*

The VeBPF many-core architecture depicted in Fig. 2 features VeBPF cores as the basic PE of this many-core architecture along with the main building block modules. The PEs of this VeBPF many-core architecture are accessed through different shared data buses for writing network packet data as PE data-memory and eBPF rules as PE program-memory. The PEs are also controlled and monitored through different control buses for various control operations like re-programming a certain PE, monitoring if any PE has finished processing by checking *Halt_out* output port and if it is HIGH then reading the result from its *R0* register along with *Error_out* and *Ticks_out* output ports, executing a certain eBPF rule on a PE selected through arbitration, etc. All of these shared data and control buses are operated by the different building blocks of this VeBPF many-core architecture that we refer to as modules.

All the building blocks of the proposed VeBPF many-core architecture including the PEs and modules, are highly flexible and can be adjusted for various target deployments, e.g., if a high performance deployment is needed like a SmartNIC, then the modules can be configured for that, similarly they can also be configured for low-resource deployment like IoT.

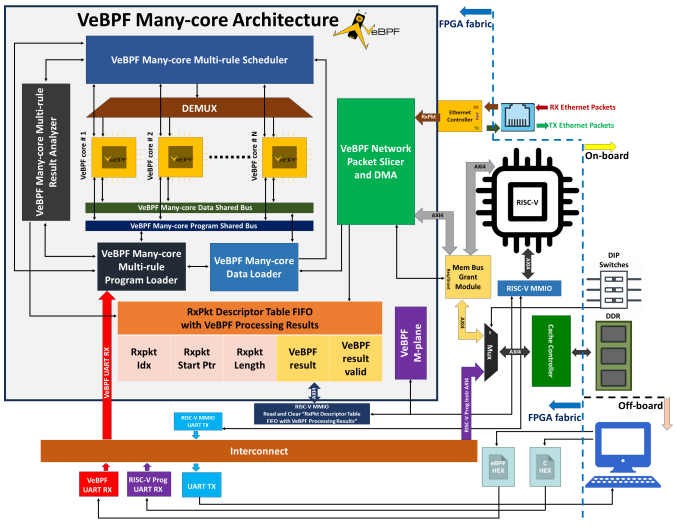


Fig. 2: Overview of the VeBPF many-core architecture.

The proposed VeBPF many-core architecture sits between the network subsystem and the memory subsystem as shown in Fig. 2. The RISC-V soft-processor controls the m-plane,

i.e., the CSRs (control status registers) of the VeBPF many-core architecture. The complex interactions of RISC-V with the VeBPF many-core architecture on the m-plane are available in the C libraries we provide with the rest of the open-source code for this VeBPF many-core architecture.

The VeBPF many-core architecture is connected with the UART subsystem as well. You can see in Fig. 2 that the VeBPF many-core architecture and the RISC-V soft-processor are sharing the same DDR memory, it is because the target deployment for the VeBPF many-core architecture shown in this Fig. 2 is for the FPGA-based IoT device deployment for which we have performed various experiments for this paper. If a different deployment was targeted like a FPGA-based SmartNIC, the VeBPF many-core architecture would have had its own separate memory to write packets to, whereas the RISC-V would have had its own separate memory. The aim of showing this particular shared memory configuration is to show the flexibility of our VeBPF many-core architecture and its ability to provide native eBPF rule-based network packet processing functionality to a resource limited FPGA-based IoT device as well as for high-end FPGA-based SmartNICs.

Before going into the details of the main modules of the VeBPF many-core architecture, a summary of the main steps required to activate the VeBPF many-core architecture is listed below:

1) The number of VeBPF cores required just need to be specified in the $N_{VeBPF}$ parameter before compilation of the FPGA bitstream. More VeBPF cores would result in faster processing of the eBPF rules;

2) The eBPF rules are uploaded from the host PC through the UART subsystem (*VeBPF UART RX*) as shown in Fig. 2. Any number of eBPF rules can be uploaded and the eBPF rule-set can be changed dynamically during run-time.

3) The RISC-V program instructions for controlling the m-plane of the VeBPF many-core architecture are uploaded through the UART subsystem as well (*RISC-V Prog UART RX*) as shown in Fig. 2. After RISC-V is activated, as a part of the m-plane functionality, it allocates the starting memory address and the total memory available for the network packets to their respective CSRs in the VeBPF many-core architecture through the MMIO bus connection. This step arms the VeBPF many-core architecture to start receiving network packets from the network subsystem;

Rest of the steps of operations are mentioned in the modules definitions of the VeBPF many-core architecture below.

### C. VeBPF Network Packet Slicer and DMA

The VeBPF network packet slice and DMA (Direct Memory Access) module is shown in detail in Fig. 3. This module is responsible for receiving the network packets from the network subsystem (*Ethernet Controller* block) through an AXI-stream port. Unlike other network packet processing frameworks [3], [14] that have to hold the network packets till processing on those packets is completed, the *Network Packet Slicer* slices the header of the network packet (*RxPkt*) and copies the network packet header and its header length in two FIFOs while handing-off the full network packet to the *DMA RxPkt to DDR* block. It is important to note that the header length varies according to the type of network packet received and users can set a custom header length as well.

The *DMA RxPkt to DDR* block checks the CSRs for the starting memory address and total memory available for the network packets. The *DMA RxPkt to DDR* block writes the *RxPkt* to memory after acquiring the grant for the memory bus from the *Mem Bus Grant Module* block and writes the corresponding *RxPkt* metadata to the *RxPkt Descriptor Table FIFO* and updates its own registers that keep track of of the currently available memory and the current memory address for the next *RxPkt* to write to. The RISC-V using the m-plane is able to read the *RxPkt* metadata from the *RxPkt Descriptor Table FIFO*. The RISC-V can then access the *RxPkts* from memory if needed, using that metadata. The RISC-V through the m-plane then clears the *RxPkt* metadata *RxPkt Descriptor Table FIFO* entries by incrementing the relevant FIFO read pointers, which signals to the *DMA RxPkt to DDR* block to increment its currently available memory. The VeBPF Network Packet Slicer and DMA module sends a HIGH *RxPktHdr_available_flag* signal to the VeBPF many-core data loader module (Fig. 4) as soon as a *RxPkt* header becomes available in the *RxPkt Headers FIFO*.

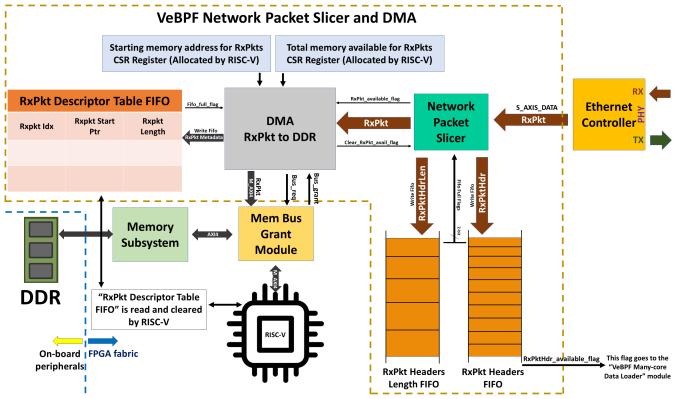


Fig. 3: VeBPF Network Packet Slicer and DMA Module

### D. VeBPF Many-core Data Loader

The VeBPF many-core data loader module reads the *RxPkt* header and its length from the FIFOs, that were written to by the VeBPF network packet slicer and DMA module, as soon as it receives a HIGH *RxPktHdr_available_flag*. The VeBPF many-core data loader loads the *RxPkt* header in all the VeBPF cores through the *VeBPF Many-core Data Shared Bus* as shown in Fig. 4.

For each 64-bit data word of the *RxPkt* header that is written to all the VeBPF cores, the *VeBPF Many-core Data Loader* block waits for acknowledgements (ACKs) from all the VeBPF cores using a reduction bit-wise AND operation on the ACK signals from all VeBPF cores, before the *VeBPF Many-core Data Loader* block writes the next 64-bit data word of the *RxPkt* header till the full header is written to all the VeBPF

cores data memories. The reason for writing the same *RxPkt* header in all the VeBPF cores is that multiple eBPF rules can be executed in-parallel on different VeBPF cores on the same *RxPkt* header so that a valid result can be obtained in minimal time, in order to keep the processing latency to a minimum.

After the VeBPF many-core data loader module uploads a *RxPkt* header to all the VeBPF cores, it sends a HIGH *VeBPF_data_loading_done_flag* signal to VeBPF many-core multi-rule scheduler module, indicating that there is a *RxPkt* header available for executing eBPF rules on.

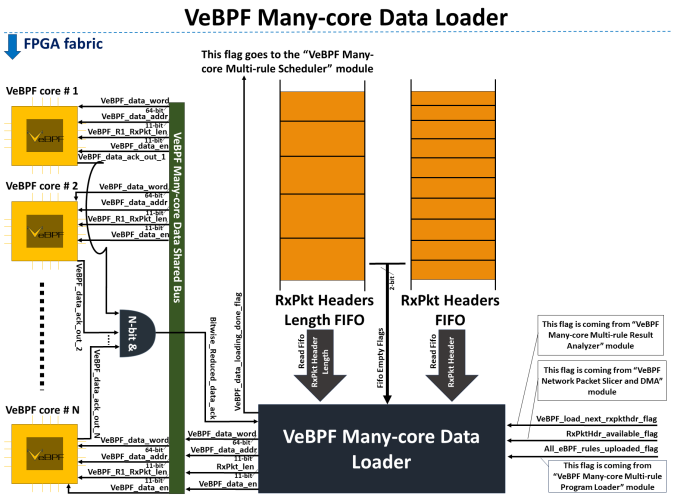


Fig. 4: VeBPF Many-core Data Loader Module

### E. VeBPF Many-core Multi-rule Program Loader

The VeBPF many-core multi-rule program loader module is what gives the VeBPF many-core architecture the flexibility to upload eBPF rules of varying lengths and the dynamicity to change the eBPF rule-set during run-time without requiring a new FPGA bitstream.

The VeBPF many-core multi-rule program loader module is depicted in Fig. 5. The eBPF rules uploaded enter from the UART subsystem (*VeBPF UART RX*) and since the eBPF rules can be of any length, the *Dynamic eBPF Rules Metadata Parser* block stores these eBPF rules in a *eBPF Rules FIFO* along with the metadata of those corresponding eBPF rules in the *eBPF Rules Metadata Table FIFO*.

Once all eBPF rules have been uploaded by the user, the *VeBPF Multi-core Multi-rule Instruction Uploader* block reads these eBPF rules and their metadata from the corresponding FIFOs and these eBPF rules are uploaded as program memory to all VeBPF cores through a shared *VeBPF Many-core Program Shared Bus*. The ACKs from all the VeBPF cores are bit-wise reduced AND-ed together after each 64-bit eBPF instruction program word is written to every VeBPF core, and this process repeats till all eBPF rules are present in every VeBPF core program memory. The reason for all VeBPF cores having all the eBPF rules in their program memory is that the VeBPF many-core multi-rule scheduler module Fig. 6 can switch to any eBPF rule on any VeBPF core in a single clock cycle by just changing the instruction pointer of that VeBPF core.

If a new eBPF rule-set is required, a HIGH *VeBPF_rst_new_rules_flag* signal is sent (Fig. 5) and the whole eBPF rules uploading process is repeated again. Once all eBPF rules have been uploaded to all VeBPF cores, the VeBPF many-core multi-rule module sends a HIGH *All_eBPF_rules_uploaded_flag* signal to the VeBPF many-core data loader module and the VeBPF many-core multi-rule scheduler module and the flag *All_eBPF_rules_uploaded_flag* is kept HIGH unless a new eBPF rule-set is being uploaded.

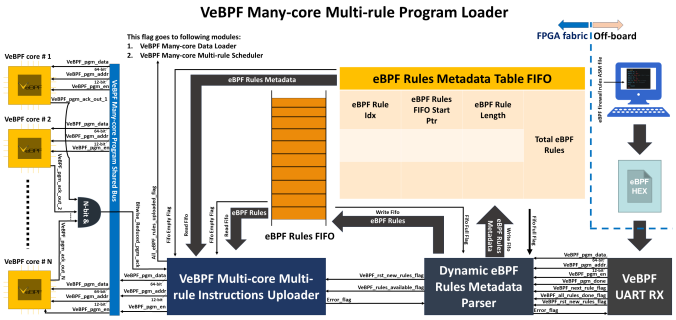


Fig. 5: VeBPF many-core multi-rule program loader module

### F. VeBPF Many-core Multi-rule Scheduler

This VeBPF many-core multi-rule scheduler module depicted in Fig. 6 is one of the most complex hardware logic in the VeBPF many-core architecture that really makes this architecture highly parallelized, resource-efficient and optimized for low latency multi-rule eBPF network packet processing.

The VeBPF many-core multi-rule scheduler module waits for HIGH flags from VeBPF many-core data loader module (*VeBPF_data_loading_done_flag*) and VeBPF many-core multi-rule program loader module (*All_eBPF_rules_uploaded_flag*), HIGH values of these flags indicate that a *RxPkt* header is available for processing in all VeBPF cores and all eBPF rules have been uploaded in all VeBPF cores respectively.

The VeBPF many-core multi-rule scheduler then checks the *VeBPF multi-core Arbitrer* block to see if any VeBPF core is idle, and as soon as the arbitrer provides an idle VeBPF core to the *VeBPF Many-core Core-Selector and Multi-rule Re-programmer* block, it reprograms that idle VeBPF core to the current eBPF rule, that hasn't been executed yet, using the eBPF rule metadata from the *eBPF Rules Metadata Table FIFO* that was filled by the VeBPF many-core multi-rule program loader module. The idle VeBPF core is reprogrammed to this selected eBPF rule in a single clock cycle by first selecting that VeBPF core through a DEMUX using the grant id given by the *VeBPF multi-core Arbitrer* as the select line of the DEMUX as shown in Fig. 6.

After the idle VeBPF core has been accessed through the DEMUX, it is reprogrammed to the selected eBPF rule in a single clock cycle by changing its instruction pointer

externally to the location of the selected eBPF rule. After the *VeBPF Many-core Core-Selector and Multi-rule Re-programmer* block reprograms the idle VeBPF core, it transfers the grant id and the relevant eBPF rules information like total eBPF rules already reprogrammed, to the *VeBPF Many-core Tracker and Rules-runner* block which basically activates/runs the reprogrammed idle VeBPF core and keeps a track of it until it reaches a *halt* state.

As soon as the VeBPF cores reach *halt* states, the *VeBPF Many-core Tracker and Rules-runner* block sends the VeBPF *R0* result information and the eBPF rules information, like total eBPF rules already reprogrammed, to the VeBPF many-core multi-rule result analyzer module (Fig.7). While the *VeBPF Many-core Tracker and Rules-runner* block is running and keeping track of all the VeBPF cores in parallel and forwarding their results to the VeBPF many-core multi-rule result analyzer module as soon as the results are received, the *VeBPF Many-core Core-Selector and Multi-rule Re-programmer* block is incrementing the eBPF rule index as soon as it hands off a reprogrammed VeBPF core to the *VeBPF Many-core Tracker and Rules-runner* block, and then *VeBPF Many-core Core-Selector and Multi-rule Re-programmer* block waits for the *VeBPF multi-core Arbitrer* block to give it the next idle VeBPF core so that the next eBPF rule can be reprogrammed on the granted idle VeBPF core and handed off to the *VeBPF Many-core Tracker and Rules-runner* block. All of these hand-shakes between *VeBPF Many-core Core-Selector and Multi-rule Re-programmer* and *VeBPF Many-core Tracker and Rules-runner* blocks are happening in parallel till the VeBPF many-core multi-rule result analyzer module sends a valid result received flag (*VeBPF_result_registered_flag*).

After the VeBPF many-core multi-rule scheduler module receives the valid result flag *VeBPF_result_registered_flag* from the VeBPF many-core multi-rule result analyzer module, it waits for the next *RxPkt* header to be uploaded by the VeBPF many-core data loader module and this whole process repeats that involves reprogramming and running eBPF rules on the multiple VeBPF cores in parallel till a valid eBPF packet processing result is received.

### G. VeBPF Many-core Multi-rule Result Analyzer

The VeBPF many-core multi-rule result analyzer module is depicted in Fig. 7 and this module takes in inputs from the VeBPF many-core multi-rule scheduler module and these inputs include the most recent eBPF processing result *VeBPF_core_most_recent_result_r0* along with total eBPF rules *Total_eBPF_rules* and total eBPF rules that have been run/reprogrammed currently *Total_eBPF_rules_reprogrammed*.

The data flow diagram in Fig. 7 tells us how VeBPF many-core multi-rule result analyzer module works. If a valid eBPF result (valid result is either ”store result”, ”error”, ”drop packet”) is received in the *VeBPF Result Analyzer* block or if the eBPF result is ”don’t care” but the *Total_eBPF_rules_reprogrammed* is equal to *Total_eBPF_rules*, then forward the eBPF processing result to the *Write VeBPF Result* block.

The *Write VeBPF Result* block appends the VeBPF result to the *RxPkt Descriptor Table FIFO* which now is renamed as *RxPkt Descriptor Table FIFO with VeBPF Processing Results*. The RISC-V soft-processor through the m-plane, can read the VeBPF processing results directly from this *RxPkt Descriptor Table FIFO with VeBPF Processing Results* table instead of having to read the *RxPkt* metadata from this table and using this metadata, read the *RxPkt* from memory and processing the packet as per the eBPF rules for the same result.

The *Write VeBPF Result* block after writing the VeBPF results sends HIGH *VeBPF_result_registered_flag* signal to VeBPF many-core multi-rule scheduler module so it can start processing eBPF rules on the next *RxPkt* header. *Write VeBPF Result* block also sends a HIGH *VeBPF_load_next_rxpkthdr_flag* signal to the VeBPF many-core data loader module signalling it to load the next *RxPkt* header into the VeBPF cores data memory.

As seen in the detailed description of the hardware architecture of the VeBPF many-core architecture modules, the limited space in this paper isn’t enough to highlight every important detail, so the detailed figures along with looking at our open-source code for this VeBPF many-core architecture would be useful for further insights. The C code libraries for the RISC-V control of the m-plane of the VeBPF many-core architecture are also an important part of the body of knowledge of this VeBPF many-core architecture.

### H. Automatic Testing & Simulation Framework

We have developed two separate automatic-testing and simulation frameworks (Fig. 8) for the further development and optimization of VeBPF CPU cores and the VeBPF many-core architecture respectively. Both automatic-testing and simulation frameworks have been developed using open-source tools like Python, Cocotb [25] and Icarus Verilog. Open-source tools like Python make it easy to simulate complex network packet interactions with our VeBPF many-core architecture. These automatic-testing and simulation tools are available on our publicly available code repositories and since they use open-source tools, anyone can use them. The VeBPF CPU core automatic-testing framework tests all the eBPF instructions on any upgrades made to the VeBPF CPU core and notifies about failures of the VeBPF core to execute any eBPF instruction, which streamlines and accelerates the development of VeBPF CPU core upgrades and the VeBPF-many core architecture.

## III. Experiments and Results

In order to show the high-configurability and versatility of the proposed VeBPF many-core architecture, we have implemented a firewall application against malicious network cyber attacks using the proposed VeBPF many-core architecture on a resource-limited FPGA-based IoT device. We used the FPGA board Arty A7-100T [26] as the FPGA-based IoT device. For the eBPF firewall experiments, we chose to implement the state-of-the-art firewall rules being used by the technology

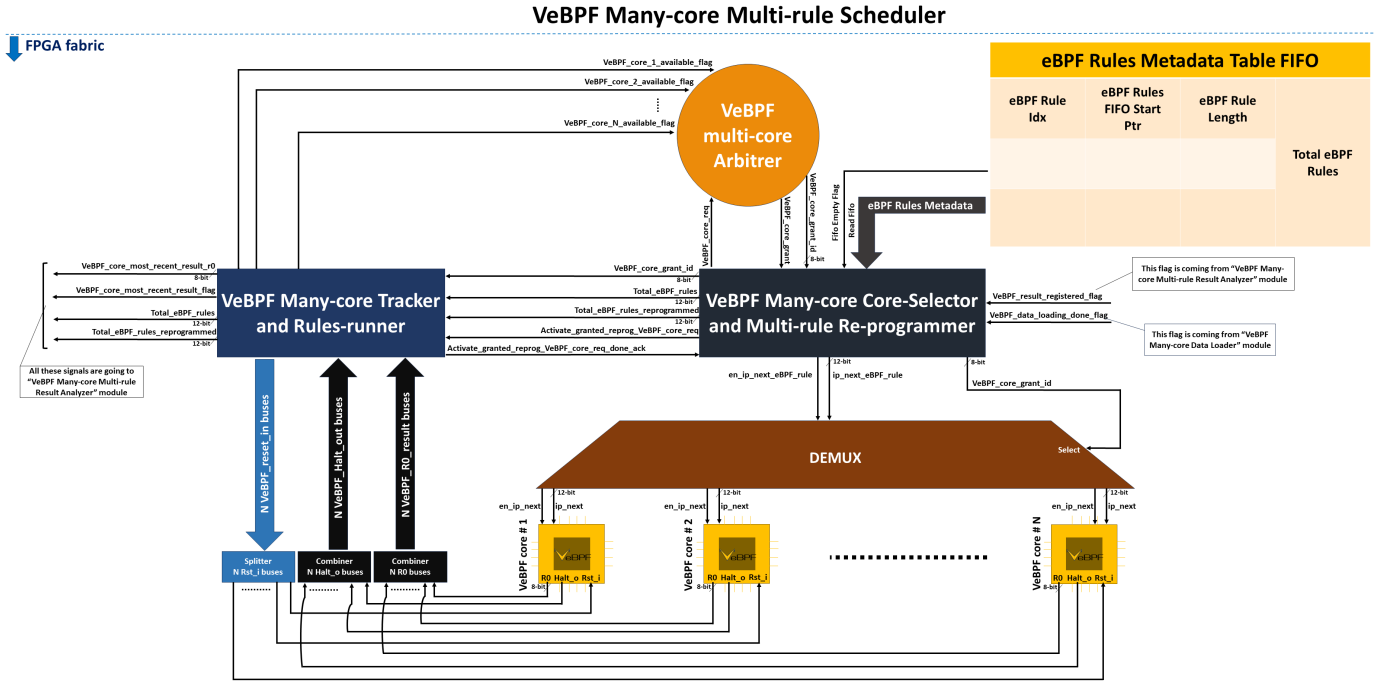


Fig. 6: VeBPF many-core multi-rule scheduler module

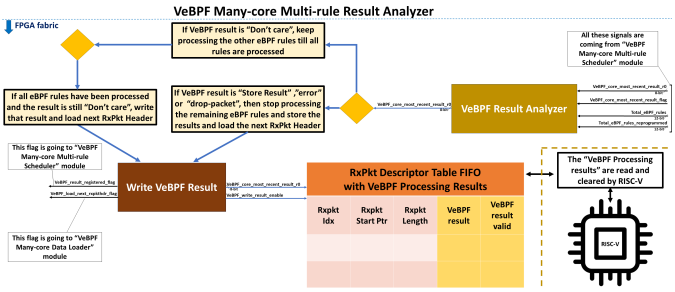


Fig. 7: VeBPF many-core multi-rule result analyzer module

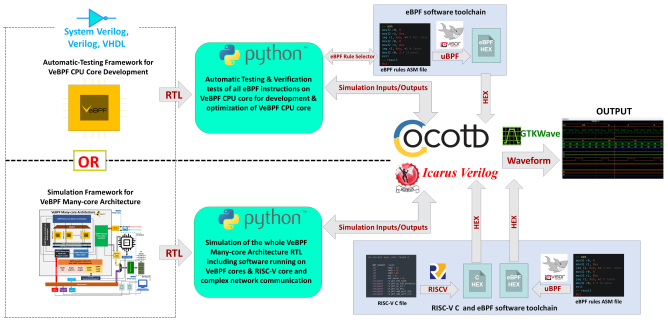


Fig. 8: VeBPF automatic-testing and simulation framework

sector as described by this technology report [27]. Table-II lists these firewall rules implemented for the experiments described below.

For evaluation of our VeBPF many-core architecture firewall application (VeBPF many-core firewall). we uploaded the Table-II firewall rules as eBPF bytecode with the total number of VeBPF cores $N_{VeBPF}$ set at 12, since it was the highest number of VeBPF cores we could synthesize along with all the required subsystems as shown in Fig. 2, before the resources ran out on the FPGA board. We evaluated the performance of the 12-core VeBPF many-core firewall by comparing its performance (latency) versus the RISC-V performance for filtering the incoming network packets according to the Table-II firewall rules. For the VeBPF many-core firewall evaluation we sent 2000 malicious *rx_pkts* conforming to each firewall rule in Table-II from the host PC at 100 Mbps to the Arty FPGA board through the ethernet port. We repeated each experiment multiple times to cater for randomization. Also, all the experiments were repeated for different sizes of the *rx_pkts* as shown in Fig. 9.

The max output firewall throughput of the 12-core VeBPF many-core firewall versus RISC-V firewall can be seen in Fig. 9 (top two graphs). It is noted here that the 12-core VeBPF many-core firewall processes the *rx_pkts* at line-rate for all packet sizes as seen by the orange line (output firewall throughput) following the blue bars (input network throughput). Whereas the RISC-V soft-processor on the FPGA-based IoT device isn't able to filter the smaller sized malicious *rx_pkts* at line-rate, even after being fully committed to filtering the network packets as per the firewall rules and not performing any other IoT functions as seen in Fig. 9 (top two graphs).

Hence the two-fold advantage that our 12-core VeBPF many-core firewall provides is:

1) Line-rate firewall filtering of incoming network packets;
2) The VeBPF firewall carries the firewall processing load,

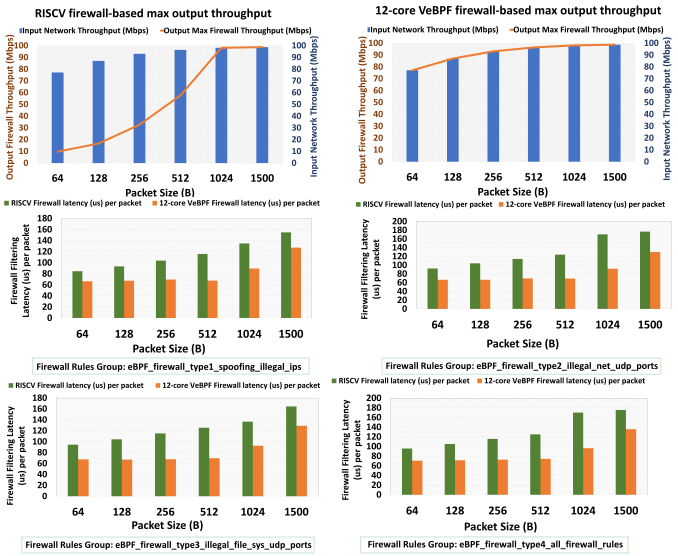


Fig. 9: Firewall throughput and latency performance comparison between VeBPF many-core firewall and RISC-V.

hence freeing up the RISC-V soft-processor to perform all the safety and mission critical IoT device related tasks;

The Fig. 9 (bottom 4 graphs) also displays the performance comparison in terms of firewall filtering latency per network packet for the 12-core VeBPF many-core firewall versus the RISC-V soft-processor for the firewall rules mentioned in Table-II. We notice a common trend that the 12-core VeBPF many-core firewall outperforms the RISC-V soft-processor for all packet sizes even after the RISC-V soft-processor is fully committed to firewall processing. The RISC-V firewall filtering results would have been way worse if the RISC-V soft-processor was performing other IoT related tasks as well, like a real IoT device, which further highlights the advantages provided by the proposed VeBPF many-core architecture for network packet processing using native eBPF bytecode.

TABLE II: Firewall Rules

| Firewall Rule Types | Blocked IPs, UDP ports |
|---|---|
| Type-1 (Illegal & Spoofing source IPs) | 255.255.255.255, 127.0.0.0, 240.0.0.0, 0.0.0.0 |
| Type-2 (Network related critical destination UDP ports) | 111, 2000, 37, 135, 137, 138, 161, 162, 514 |
| Type-3 (File system related critical destination UDP ports) | 69, 2049, 389, 4045 |
| Type-4 (All rules combined) | All 3 firewall rule types combined |

## IV. Conclusion and Future Work

In this paper we presented a VeBPF many-core architecture that provides network packet processing functionality for both high-end and low-end FPGAs for FPGA target deployments like SmartNICs and IoT. This VeBPF many-core architecture is built using VeBPF CPU cores developed by us as the PE of this many-core architecture. These VeBPF cores are eBPF ISA compliant and are specialized for network packet processing and use native eBPF bytecode as program memory.

Our experimentation on the FPGA-based IoT device as the target deployment of the proposed VeBPF many-core architecture shows that the VeBPF many-core firewall implements state-of-the-art firewall rules for incoming malicious network attacks and filters them at line-rate and takes the processing load off the main IoT device processor (RISC-V) as compared to a dedicated RISC-V soft-processor which is slower.

For future work we want to show the proposed VeBPF many-core architecture implemented on a FPGA-based SmartNIC and HPC applications. We are also looking into adding more configuration options in the VeBPF many-core architecture. We have released the code, for VeBPF CPU core and VeBPF many-core architecture Verilog HDL along with their simulation frameworks built using open-source tools and the C libraries for the RSIC-V m-plane of the VeBPF many-core architecture, as an open-source contribution for further advancement of FPGAs in many-core architectures and communication.

## Acknowledgments

This work was supported, in part, by Red Hat through award 2024-01-RH08.

## References

[1] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen, "Corundum: An open-source 100-gbps NIC," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2020, pp. 38–46.

[2] K. Deierling, "What is a dpu," 2020. [Online]. Available: https://blogs.nvidia.com/blog/whats-a-dpu-data-processing-unit/

[3] J. Lin, K. Patel, B. E. Stephens, A. Sivaraman, and A. Akella, "PANIC: A High-Performance Programmable NIC for Multi-tenant Networks," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 243–259.

[4] Q. Xiong, A. Skjellum, and M. Herbordt, "Accelerating MPI Message Matching Through FPGA Offload," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 191–1914, doi: 10.1109/ FPL.2018.00039.

[5] Q. Xiong, C. Yang, R. Patel, T. Geng, A. Skjellum, and M. Herbordt, "GhostSZ: A Transparent SZ Lossy Compression Framework with FPGAs," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 258–266, doi: 10.1109/FCCM.2019.00042.

[6] A. Guo, T. Geng, Y. Zhang, P. Haghi, C. Wu, C. Tan, Y. Lin, A. Li, and M. Herbordt, "FCsN: A FPGA-Centric SmartNIC Framework for Neural Networks," in *30th IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2022, dOI: 10.1109/FCCM53951.2022.9786193.

[7] ——, "A Framework for Neural Network Inference on FPGA-Centric SmartNICs," in *International Conference on Field-Programmable Logic and Applications*, 2022, dOI: 10.1109/FPL57034.2022.00071.

[8] A. Kamaleldin, S. Hesham, and D. Göhringer, "Towards a modular RISC-V based many-core architecture for FPGA accelerators," *IEEE Access*, vol. 8, pp. 148 812–148 826, 2020.

[9] A. Guo, Y. Hao, C. Wu, P. Haghi, Z. Pan, M. Si, D. Tao, A. Li, M. Herbordt, and T. Geng, "Software-hardware co-design of heterogeneous smartnic system for recommendation models inference and training," in *ICS 2023: International Conference on Supercomputing*, 2023, dOI = 10.1145/3577193.3593724.

[10] J. Gray, "GRVI Phalanx: A massively parallel RISC-V FPGA accelerator accelerator," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2016, pp. 17–20.

[11] A. Kurth, P. Vogel, A. Capotondi, A. Marongiu, and L. Benini, "HERO: Heterogeneous embedded research platform for exploring RISC-V manycore accelerators on FPGA," *arXiv preprint arXiv:1712.06497*, 2017.

[12] J. Schulist, D. Borkmann, and A. Starovoitov, "Linux Socket Filtering aka Berkeley Packet Filter (BPF)," March 17, 2019. [Online]. Available: www.kernel.org/doc/Documentation/networking/filter.txt

[13] eBPF, "eBPF Case Studies." [Online]. Available: https://ebpf.io/case-studies/

[14] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco, "hXDP: Efficient software packet processing on FPGA NICs," *Communications of the ACM*, vol. 65, no. 8, pp. 92–100, 2022.
[15] R. Prinz, "hbpf." [Online]. Available: https://github.com/rprinz08/hBPF
[16] S. Bandara, A. Sanaullah, Z. Tahir, U. Drepper, and M. Herbordt, "Enabling VirtIO Driver Support on FPGAs," in *8th International Workshop on Heterogeneous High Performance Reconfigurable Computing*, 2022, doi: 10.1109/H2RC56700.2022.00006.
[17] ——, "Performance Evaluation of VirtIO Device Drivers for Host-FPGA PCIe Communication," in *31st Reconfigurable Architectures Workshop (RAW)*, 2024, doi: 10.1109/IPDPSW63119.2024.00043.
[18] S. Bandara, N. Cherry, and M. Herbordt, "Fully Transparent Client-Side Caching for Key-Value Store Applications Using FPGAs," in *IEEE High Performance Extreme Computing Conference*, 2024.
[19] M-labs, "A python toolbox for building complex digital hardware," 2015. [Online]. Available: https://github.com/m-labs/migen
[20] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung *et al.*, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 51–66.
[21] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry, "Achieving 100gbps intrusion prevention on a single server," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 1083–1100.
[22] C. Yang, "FPGA in IoT Edge Computing and Intelligence Transportation Applications," in *2021 IEEE International Conference on Robotics, Automation and Artificial Intelligence (RAAI)*. IEEE, 2021, pp. 78–82.
[23] L. Catarinucci, D. De Donno, L. Mainetti, L. Palano, L. Patrono, M. L. Stefanizzi, and L. Tarricone, "An IoT-aware architecture for smart healthcare systems," *IEEE internet of things journal*, vol. 2, no. 6, pp. 515–526, 2015.
[24] A. D. Wood and J. A. Stankovic, "Denial of service in sensor networks," *Computer*, vol. 35, no. 10, pp. 54–62, 2002.
[25] Cocotb, "Cocotb, a coroutine based cosimulation library for writing vhdl and verilog testbenches in python." [Online]. Available: https://github.com/cocotb/cocotb
[26] Digilent, "Arty A7-100T: Artix-7 FPGA Development Board." [Online]. Available: https://digilent.com/reference/programmable-logic/arty-a7/reference-manual
[27] SANS Institute, "Security consensus operational readiness evaluation firewall checklist." [Online]. Available: https://www.sans.org/media/score/checklists/FirewallChecklist.pdf