Faster Software Packet Processing on FPGA NICs with eBPF Program Warping

Marco Bonola, *CNIT/Axbryd;* Giacomo Belocchi, Angelo Tulumello, and Marco Spaziani Brunella, *Axbryd/University of Rome Tor Vergata;* Giuseppe Siracusano, *NEC Laboratories Europe;* Giuseppe Bianchi, *University of Rome Tor Vergata;* Roberto Bifulco, *NEC Laboratories Europe*

https://www.usenix.org/conference/atc22/presentation/bonola

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

Faster Software Packet Processing on FPGA NICs with eBPF Program Warping

Marco Bonola[1,2], Giacomo Belocchi[1,3], Angelo Tulumello[1,3], Marco Spaziani Brunella[1,3], Giuseppe Siracusano[4], Giuseppe Bianchi[3] and Roberto Bifulco[4]

[1]Axbryd, [2]CNIT, [3]University of Rome Tor Vergata, [4]NEC Laboratories Europe

Abstract

FPGA NICs can improve packet processing performance, however, programming them is difficult, and existing solutions to enable software packet processing on FPGA either provide limited packet processing speed, or require changes to programs and to their development/deployment life cycle.

We address the issue with *program warping*, a new technique that improves throughput replacing several instructions of a packet processing program with an equivalent runtime programmable hardware implementation. Program warping performs static analysis of a packet processing program, described with Linux's eBPF, to identify subsets of the program that can be implemented by an optimized FPGA pipeline, the *warp engine*. Packets handled by the warp engine are eventually delivered to a regular eBPF program executor, along with their program context (registers, stack), to complete execution of those program parts that cannot be efficiently pipelined.

We prototype program warping on a 100Gbps FPGA NIC, extending hXDP, a state-of-the-art eBPF processor for FPGA, and measure its performance running 6 unmodified real-world eBPF programs, including deployed applications such as Katran and Suricata. Our prototype runs at 250MHz, uses less than 15% of the FPGA resources, and improves hXDP throughput by 1.2-3x in most cases, and up to 18x.

1 Introduction

Datacenter and telecom operators deploy FPGA NICs to handle network port speeds of 100Gbps or more, and to support heterogeneous applications and workloads [12, 21, 30]. In fact, these devices can host multiple accelerators [23], e.g., for radio signal processing, therefore providing a common hardware platform to address different scenarios [11].

Nonetheless, for network packet processing functions, such as firewalls or load balancers, FPGA NICs raise several challenges. First, programming FPGAs is difficult, often requiring dedicated teams of hardware specialists [15]. Second, it involves longer synthesis-implementation cycles that may take hours to complete, before a new program version can be finally deployed. This is at odds with current practices that foster continuous deployment cycles, and frequent updates to the packet processing programs [1, 7]. Finally, packet processing functions should consume only minimal FPGA hardware resources, to leave space to other accelerators required for signal processing, machine learning, etc.

These requirements, when combined, rule out existing solutions that cannot dynamically change the implemented packet processing programs, such as those based on the high-level synthesis of programs described with domain-specific languages [3, 38, 40] like P4 [8]. Alternative approaches, such as hXDP [10], explicitly address these challenges, but at the cost of a lower packet forwarding throughput. In fact, hXDP implements on the FPGA a processor-based executor for network programs described with eBPF, which provides a remarkable but still limited throughput performance, comparable to that of a single multi-GHz CPU's core [10].

Our goal is to improve on this figure, and significantly increase throughput while respecting the listed requirements. We follow the conceptual approach of hXDP, embracing the Linux's eBPF framework and its programming model to describe packet processing functions. However, we introduce a new technique, *program warping*, which leverages common properties of eBPF programs to automatically replace the execution of many program's instructions with a semantically equivalent hardware-supported implementation, thereby reducing the program execution time and increasing throughput. As we will see, in some real-world use cases our system can achieve up to an 18x higher throughput than hXDP.

Program warping builds on the observation that a subset of the eBPF programs' instructions implement common packet processing tasks, such as packet header parsing, which can be efficiently implemented in pipeline-parallel architectures [9, 13]. Therefore, under the constraint of keeping transparency to the programmer, we address two main issues in our design: (i) identifying, from the eBPF program's bytecode, the tasks that can be efficiently parallelized in a pipeline; (ii) designing an FPGA pipeline that runs such tasks, while providing runtime

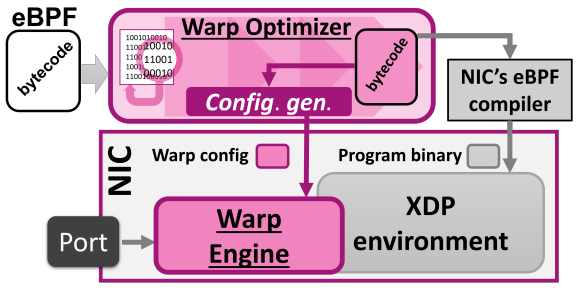


Figure 1: **Program warping includes two components: an optimizing compiler and a hardware engine for FPGAs**

reconfigurability and using minimal hardware resources.

We prototype program warping extending hXDP with a new compiler, the *Warp Optimizer*, and a new hardware module, the *Warp Engine* (cf. Figure 1). The Warp Optimizer performs static analyses of the eBPF bytecode, leveraging the eBPF's machine model to make several simplifying assumptions, e.g., about memory areas' content, in order to infer the program's intent. In this step, the Warp Optimizer identifies set of instructions whose execution can be performed by the Warp Engine. The Warp Engine is integrated with the hXDP's eBPF executor. Packets received by the system are processed by the Warp Engine first, and then passed to the eBPF executor to run the subset of the program that cannot be accelerated. In turn, this allows us to streamline the Warp Engine implementation, which resembles a fused parser plus match-action pipeline, supporting only the minimal set of functionality required to accelerate common packet processing tasks, and leaving to the eBPF executor any more complex functionality.

We evaluate our system with a 100Gbps Xilinx Alveo U50, running 6 real-world use cases: the IP router and the tunneling examples from the Linux XDP's application examples; a Dynamic NAT; the Facebook load balancer Katran [14]; and the Network Security Appliance Suricata [39]. On this set of applications and compared to hXDP, program warping provides a per-application speed-up of at least 1.2-3x, and up to 18x, while running at 250MHz and keeping the overall FPGA resources occupation below 15%. To put this result in perspective, running Suricata in software (Linux v.5.4.0) on a single Intel Xeon 4410 CPU core achieves a throughput of about 8.7 Million packets per second (Mpps), whereas our program warping prototype achieves up to 83Mpps in a similar setting. This shows that program warping is beneficial in all cases, but clearer advantages appear with programs that have a heavier packet parsing and classification component. In summary, we contribute:

- A method to extract high-level packet processing tasks from eBPF bytecode, and generate functionally equivalent descriptions that combine parsing descriptions, match-action rules and subsets of the original bytecode;
- A hardware extension to hXDP, which implements high-performance and runtime-configurable packet data reading and classification using few FPGA resources;
- The evaluation of the end-to-end system using 6 real-world use cases and extensive micro-benchmarks.

2 Goal, Requirements and Challenges

Our goal is to provide a significant increase (i.e., >2x) to the throughput of eBPF packet processing programs on FPGA NICs, while meeting the following requirements:

1. The system should run unmodified eBPF programs
2. The system should support dynamic program loading
3. The system should use a small fraction of the FPGA resources (<20%)

This is challenging for several reasons. First, previous work like hXDP already explored the optimization space for eBPF programs on FPGA, leveraging instruction-set specialization and instruction-level parallelism to reduce the number of programs' instructions and run them in parallel, when possible. This suggests that additional instruction-level optimization is unlikely to provide large gains. Second, our solution space is limited since we cannot change the eBPF programming model. For instance, we cannot pursue any solution that would require programmers to annotate their code, e.g., to discover parallelization opportunities. Third, the need to support dynamic program loading rules out approaches that implement programs as hardware pipelines, e.g., like in Emu [38] and P4->NetFPGA [3]. Finally, the requirement to use little FPGA resources makes effectively impossible to use any solution that requires complex logic implementation in hardware. For reference, even streamlined hardware designs like hXDP already consume about 10% of the FPGA resources.

Non-goal Since we use only a fraction of the FPGA for network packet processing, we do not have the goal of matching the throughput of designs entirely dedicated to the task.

3 Concept and Background

In this Section we provide background about eBPF/XDP, and then present the program warping concept and system design.

3.1 Background: eBPF and XDP

eBPF is a Linux technology used to implement load balancing [14], security [7], monitoring [2], deep packet inspection [5], policy enforcement [1], and more.

The eBPF framework runs small programs within the Linux kernel using a virtual machine (VM) with its own Instruction Set Architecture (ISA). The VM implements a register architecture, with a program counter (PC), 10 general registers ($R0 - R9$), and a read-only stack pointer ($R10$) that contains the address of a 512B memory area used as program's stack. These capture the current program's state, which resets for every new run. eBPF provides *maps* data structures to save state across program runs. These are memory areas defined at compile time and organized as lookup tables.

eBPF programs written in a high-level language, such as C, are compiled to the eBPF bytecode. The eBPF bytecode can be loaded in the kernel using different *hooks*. We focus on the XDP hook [19], and call *XDP programs* an eBPF program attached to the XDP hook. The hook is provided at the NIC driver level. When a packet is received, the XDP environment: (i) creates an `xdp_md` struct to contain the packet buffer pointers and metadata, such as the packet's input port id; (ii) sets *R*0 to point to the address of the memory area hosting the struct; (iii) and then starts the VM to run the XDP program. At the end of its execution, the program can return a forwarding decision for the packet by writing the forwarding action code in *R*0.

When loaded in the Linux kernel, the bytecode is statically verified to ensure safety, e.g., guaranteed program termination. To enable verification, eBPF programs can only use a subset of the C expressive power. For instance, unbounded cycles and dynamic memory allocations are not allowed. To finally run on the target hardware, a second (just-in-time) compilation step translates the eBPF bytecode to the target machine code.

Program example Listing 1 shows an XDP program written in C. The program checks if the source MAC address of IPv4 packets is in a hashtable. If so, it passes the packet to the Linux's network stack. Otherwise, the packet gets dropped. Furthermore, the program drops any IPv6 packet, and passes to the network stack any packet that is neither IPv4 or IPv6.

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

```
int l2_acl(struct xdp_md *ctx) {
 void *data_end =(void *)(long)ctx->data_end;
 void *data = (void *)(long)ctx->data;
 void *lookup_res = NULL;
 __u32 proto, nh_off;
 struct ethhdr *eth = data;
 __u8 key[6] = {0};
 nh_off = sizeof(struct ethhdr);
 if (data + nh_off > data_end) {
  return XDP_DROP;
 }
 proto = eth->h_proto;
 if (proto == BE_ETH_P_IP) {
  __builtin_memcpy(key, eth->h_source, 6);
  entry = bpf_map_lookup_elem(&map, &key);
  if (entry) {
   return XDP_PASS;
  } else {
   return XDP_DROP;
  }
 } else if (proto == BE_ETH_P_IPV6) {
  return XDP_DROP;
 } else {
  return XDP_PASS;
 }
}
```

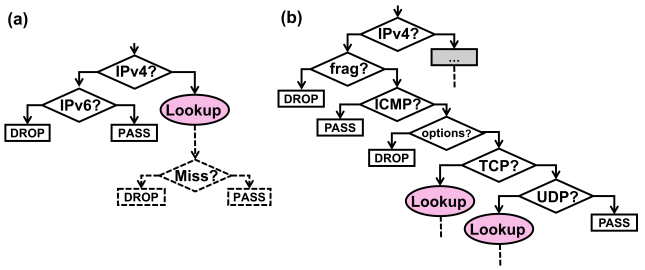


Figure 2: **Flow diagrams for the program from Listing 1 (a) and for (part of) Katran (b). The solid lines show the part of the program that only needs reading packet data and comparison operations.**

3.2 Program Warping

eBPF executors on FPGA have limited throughput when they need to process many eBPF instructions: FPGA designs usually have a low clock frequency (e.g., <400 MHz), making running an instruction expensive. While parallel instructions execution is possible, the level of parallelization is at most 2-3 instructions per clock cycle [10]. Reducing the number of instructions can increase throughput, but *can we achieve functional equivalence with less instructions?*

To answer the question, we studied several XDP programs. We show two examples in Figure 2, where we report the control flow for the program from Listing 1 (a), and for (a subset of) Katran [14] (b), an XDP L4 load balancer deployed in production by Facebook. In all the studied cases, the first part of the program has instructions that only perform reading from the packet data and comparisons with constants. This is the case since network packet processing programs usually perform packet header parsing and classification as a first step. After that, the programs diverge significantly, with operations that are specific to the application logic.

For a hardware implementation, read only access to a single (small) memory means no data hazards to handle (i.e., no read/write conflicts), and that hardware wires routing may be simple, since a single memory contains all the needed data. In fact, the operations of the first program's part may be described by a few match-action rules, as shown in Table 1 for Listing 1. Rules #1 and #3 only read some bits from the packet and check their value. Rule #2 is more complex, since it requires the execution of the program's part that includes the lookup and its downstream operations. More generally, this second part of the program requires to read/write memory areas beyond that containing the packet data.

From this observation emerges the core idea of program warping: run the first program's part on an extremely simple executor, and then use a downstream eBPF processor to run only the remaining, more complex instructions. This enables skipping several instructions, and increase throughput.

3.3 System Design

Without loss of generality, we design program warping as an extension to hXDP (Figure 1), the current state-of-the-art to

#	eth_proto	action
1	IPv6	DROP
2	IPv4	Continue Processing
3	*	PASS

Table 1: **Match-action rules to implement part of the program from Listing 1. Rule #2 needs additional processing and accessing data beyond the packet's content.**

run XDP programs on FPGA NICs [10]. hXDP provides the XDP environment for the FPGA NIC, and a compiler that takes eBPF bytecode and outputs the machine code for the on-NIC XDP environment. We inherit the eBPF programming and deployment models from hXDP: from the perspective of an eBPF/XDP programmer, program warping does not introduce any change.

Compile time When loading an XDP program's bytecode to the NIC, program warping extends the hXDP compiler by triggering a new compiler first: the *Warp Optimizer*. The Warp Optimizer performs static analysis on the bytecode to identify the instructions that can *warped*, i.e., they can be run by the Warp Engine. The output of this process is a configuration for the Warp Engine in the form of match-action rules.

Runtime At runtime, the Warp Engine receives packets first, and applies the match-action rules. If a forwarding decision can be already taken at this stage, the Warp Engine sets the value of the XDP Environment's $R0$, and transfers the packet to the XDP Environment that carries out the forwarding action. If instead the program cannot run entirely in the Warp Engine, then a context restoration is triggered. Context restoration allows the eBPF executor to skip the warped instructions, while ensuring a correct internal state to start processing the remaining instructions. This involves copying data from the packet to the XDP Environment's $R0 - R9$ and stack memory, and setting the program counter to point to the next program's instruction. The Warp Engine performs such operations in parallel while also transfering the packet to the XDP environment, where finally the processing continues to terminate the program's execution. That is, the Warp Engine and the XDP Environment work in pipeline: while the XDP Environment processes a packet, the Warp Engine is processing the next packets. This ensures that the introduction of the Warp Engine *never* reduces the system throughput, and that in the worst case it only introduces an often negligible increase (10s of nanoseconds) of the packet processing latency.

4 Warp Optimizer

The Warp Optimizer is a custom compiler that takes as input the eBPF bytecode and produces as output: (i) the set of bits that should be extracted from the packet data; (ii) a set of match-action rules that will replace the *warped* (i.e., removed) program's instruction; (iii) the description of the context associated to *context restore* actions. The rules' match conditions are described by a set of couples `(offset, length)`, which specify the bits of the packet that should be read. The actions can be of one of the following two types:

- An XDP **forwarding decision** that neither modifies the packet nor the internal state of the system (e.g., the content of the maps), i.e., `DROP`, `PASS`, `TX` or `REDIRECT`;
- A **context restore** to continue execution in the XDP Environment, configured using the provided program counter and context (i.e., registers and stack content).

Here, there are two important design decisions that allow us to minimize hardware complexity. First, the definition of the match conditions may be thought as roughly corresponding to the definition of packet header's fields, however the Warp Optimizer (and the Warp Engine) have no knowledge of what a header field is. We purposely avoided the implementation of a complete packet header parser logic [18], opting instead for a simpler set of reads of a sequence of bit vectors from the packet data. This allows us to avoid the implementation of state machines and enables a fully pipelined execution of the bit vector extraction. Second, the Warp Optimizer only provides a forwarding decision action when there is no modification to the packet and no *side effects* due to the packet processing, e.g., map accesses. Modification to the packet would require additional hardware machinery, e.g., to compute values and write them in the specific packet's positions. Instead, accessing any internal state of the system would increase hardware complexity significantly, requiring a tighter integration with the XDP environment, and introducing potential data hazards due to e.g., read-after-write for packets processed back-to-back in the Warp Engine pipeline [37].

4.1 Program analysis

To extract the Warp Engine configuration, the Warp Optimizer performs static analysis of the input program. Here, recall that XDP programs can implement arbitrary computations, which generally complicates any static analysis task [29]. Nonetheless, eBPF is designed to simplify static verification of programs loaded in the Kernel, which helps also our analysis. In particular, we benefit from the definition of three logically distinct memory areas: (i) the packet buffer; (ii) the stack; (iii) and maps. Each of these areas can be easily identified. The packet buffer is retrieved from the `struct xdp_md`, whose address is in eBPF VM's register $R1$ when a program starts. The stack base address is stored in the read-only register $R10$. Finally, maps are always accessed using a specific eBPF helper function. With this information the Warp Optimizer can trace accesses to the different memory areas, and infer the evolution of the program state.

In greater detail, the Warp Optimizer first builds the program's *Control Flow Graph* (CFG), e.g., see left part of Figure 3. The CFG is a directed graph, in which each *node* represents a code *block*, i.e., a set of instructions that are all executed if the program's control flow triggers the execution of the block's first instruction. The directed edges show how

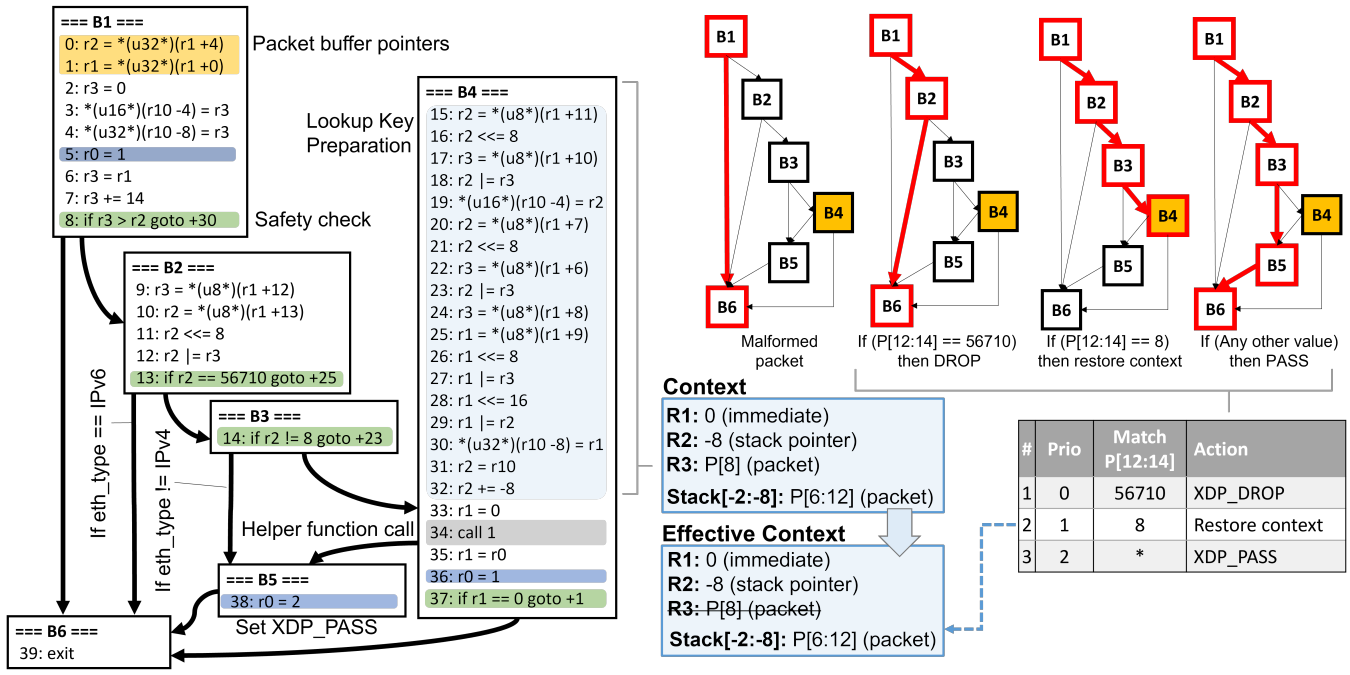


Figure 3: **The operations of the Warp Optimizer for the program from Listing 1: (i) Control Flow Graph analysis; (ii) Match-action rules extraction; (iii) Context identification. All the operations are performed at compile time.**

the different blocks might be executed one after the other, depending on the results of (conditional) jumps. Second, the Warp Optimizer converts the eBPF instructions, which use physical registers, into a Static Single Assignment (SSA) form. In this form, physical registers are substituted with variables that identify the instruction that have defined them. This helps the tracking of the values accessed by each instruction, and therefore it allows the Warp Optimizer to identify the accessed packet data, and the values stored into stack and registers.

After these two processing steps, the Warp Optimizer divides the CFG's blocks in three categories: start node; middle nodes; and terminal nodes. Terminal nodes are the blocks containing as last instruction `exit`, `call` or instructions that write to the packet data. Middle nodes are all the nodes that are not the start or terminal nodes, and they are further categorized in matching and non-matching nodes. This depends on whether the block ends with a conditional jump instruction (matching) or with any other instruction (non-matching).

4.2 Match-action rules generation

Match-action rules generated by the Warp Optimizer are triples $\langle matches, action, priority \rangle$, where `matches` is a list of (offset, length) pairs, `action` is either a forwarding decision or context restore, and `priority` is an integer value where the lower number encodes the higher priority. To generate these triples, the Warp Optimizer runs Algorithm 1. The algorithm defines a zero initialized current priority counter, a list of bitvectors extracted from the packet (`fields`), their corresponding matching values (`matches`), and the current `stack` and `registers`. Then, it performs Depth-First Search (DFS) on the CFG, descending from the start node and stopping when it reaches a terminal node. The paths explored with this way capture the part of the program that can be *warped*.

When performing DFS, the algorithm evaluates all the instructions in the node, updating the current stack and registers state (i.e., the `registers` and `stack` arrays). For each middle node, if it is a matching node, the algorithm creates a copy of the `fields` and `matches`, and adds to them the bitvector checked by the current's block matching condition. That is, the condition of the conditional jump, and the variable's value used in the condition, respectively. This corresponds to checking `if packet_data[s:e] == X`, where `[s:e]` identifies a vector of `e - s` bits in the packet starting at offset `s`, and X is an `e - s` long bitvector. The current `registers` and `stack` are also copied, since the algorithm has to explore the two branches coming after the conditional jump, for which the program's state will evolve differently. When doing so, the algorithm explores first the branch corresponding to the *jump taken* case. When completing the exploration of that branch, the algorithm comes back to the latest encountered branching point, to explore the other branch, i.e., the one corresponding to the *jump not-taken* case (see right-top part of Figure 3).

A branch exploration terminates when there is a terminal node. After evaluating the instructions in the terminal node, the algorithm creates a `match-action rule` using the current list of `matches` and the current `priority` value. To define the action associated to the rule, the algorithm looks at the nodes's last instruction. If it is an `exit` instruction the action is a forwarding decision, defined by the currently evaluated value of the `r0` register. Other-

Algorithm 1: Warp Optimizer Algorithm

```
priority ← 0
matches, fields, rules, registers, stack ← [ ]
Function get_MAT(block, matches, fields, priority, rules,
 stack, registers):
    evaluate_instructions(block, registers, stack)
    last_insn ← block.instructions[LAST]
    if is_terminal(block) then
        if is_exit(last_insn) then
            rule ← ⟨matches, Action(r0), priority⟩
        else
            action ←
              Action(PC=last_insn.pc, registers, stack)
            rule ← ⟨matches, action, priority⟩
        rules ← rules ∪ {rule}
        priority++
    else
        block+ ← block.tnext
        block− ← block.fnext
        if is_match(last_insn) then
            fields ← fields ∪ {PacketField(last_insn)}
            matches+ ← matches ∪ {Match(last_insn)}
            get_MAT(block+, matches+, priority, rules,
              stack, registers)
        get_MAT(block−, matches, priority, rules, stack,
          registers)
```

wise, the action is a restore context action, which includes $\langle pc, restored\_stack, restored\_registers \rangle$, where pc is the program counter of the instruction immediately following the node's last instruction, *restored_stack* and *restored_registers* are the evaluated current stack and registers, i.e., the *context* to be restored (right-bottom part of Figure 3). After the rule creation, the priority counter is incremented. Since the CFG is explored by selecting first the branch-taken path, this ensures that the rules having the longer match list have higher priority, which is then useful to simplify the rule matching logic implementation at runtime.

5 Warp Engine

The Warp Engine is a pipelined implementation of a *fused* packet parsing and match-action unit, in principle similar to those implemented in switching ASICs, such as RMT [9], but with important conceptual differences and simplifications that are enabled by the co-design with the Warp Optimizer. For instance, in the Warp Engine there is no distinction between the input parser and the match-action unit.

Figure 4 shows an overview of the Warp Engine architecture. We can identify three conceptual sub-systems. First, there is a *key extraction unit*, which comprises twelve stages and is in charge of building a 16B long vector extracting bits from the packet data. Second, a *match-action unit* uses the key to perform a lookup for a matching entry, which is associated with three areas in three distinct memories. These memory areas store the type of action associated with the packet, and the program context (register, stack) that should be restored in case of a context restore action. Finally, the last sub-system is the *context restoration unit*, which extracts the packet data required to build the context for a packet that needs to continue processing at the end of the Warp Engine. In our design, we use hXDP to implement the XDP environment on the FPGA, slightly modifying it to enable the Warp Engine to hook into the registers and stack memories.

An important aspect of the Warp Engine design is that the three sub-systems are part of a single pipeline that by design never stalls. In fact, the only case in which the pipeline stages do not advance processing is when hXDP is busy processing a previous packet, and therefore the hXDP's Active Packet Selector cannot host a new packet in its buffer memory. This design has also a second effect, since hXDP is still in charge of the forwarding of each and any packet, the Warp Engine has no impact on the packets ordering.

5.1 Key Extractor

The Key Extractor is connected to the packet input queue through a *splitter*, which duplicates the first 128B of the packet to forward them to the Key Extractor's pipeline. The pipeline includes 12 stages, and each of them implements a configurable extractor module. The extractor reads up to 2Bs from the duplicated packet chunk, performs a simple bitwise operation on them, e.g., `and`, with a 2B long constant value, and finally writes the result of such operation to a lookup key buffer. Which bytes to read, what operation to perform, and the value of the constant are all runtime-configurable parameters that are provided by the Warp Optimizer. Each extractor performs its operations in a single clock cycle, and passes to the next extractor: (i) its modified lookup key buffer; (ii) the offset at which to write in such buffer; (iii) the packet chunk.

5.2 Match-action Unit

The match-action includes a ternary-addressable content memory (TCAM), and three memory areas: (i) the *Action Memory*, to store the actions associated to the TCAM entries, including action type, *R*0 value and program counter; (ii) *Registers Configuration Memory*, which stores the Context Restoration Unit's configuration to extract the register values; (iii) *Stack Configuration Memory*, which provides a similar configuration but related to the stack. These three areas are organized in *lines* of different sizes, and for each memory the number of lines is equal to the maximum number of TCAM entries. The lookup key provided by the key extractor is used to find the matching entry in the TCAM, which is associated to a single *line number* that is then used to access in parallel the three memory areas. If the line extracted from the action's memory includes an action of type forwarding decision, then the pipeline propagates only such line to the next stage, to

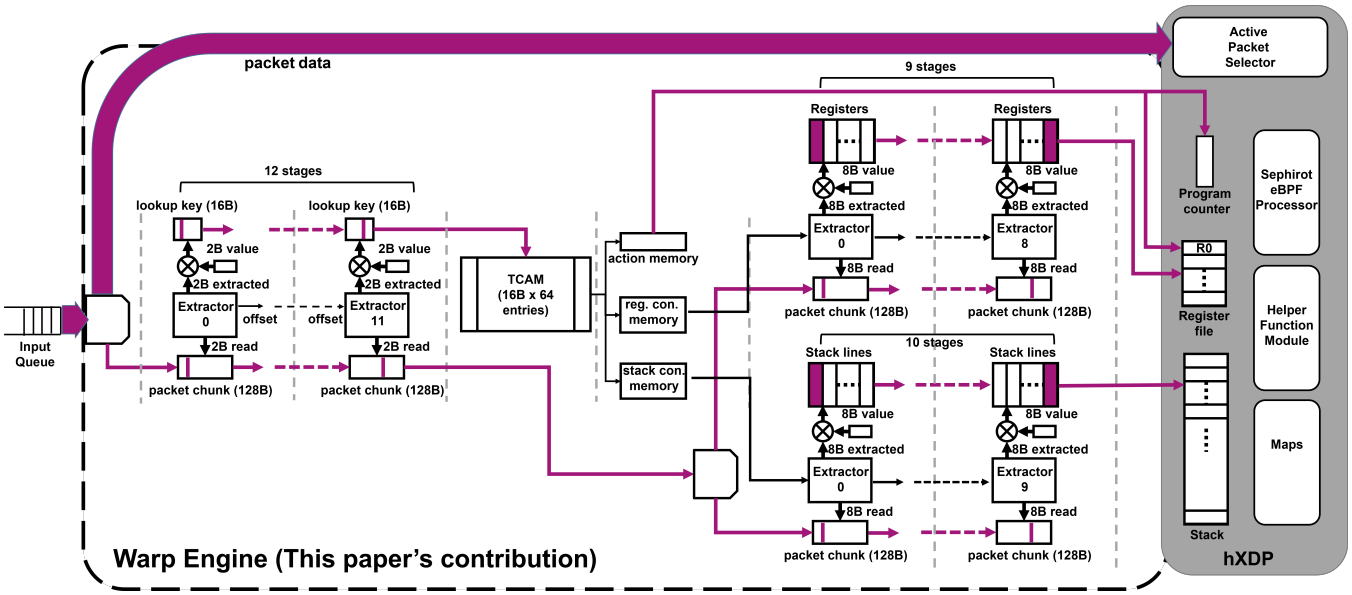


Figure 4: **Warp Engine's architecture. The pipeline stalls only when hXDP is not ready to receive a next packet.**

eventually configure the XDP Environment. Otherwise, each of the three lines extracted from the three memories is provided to the downstream pipeline. The memory lines contain the full configuration for the Context Restoration Unit, in order to build the stack and registers values. Both the TCAM entries and the memory lines are configured at runtime with the output of the Warp Optimizer.

5.3 Context Restoration Unit

The Context Restoration Unit comprises two parallel pipelines, which are composed of modules closely resembling the Extractor modules of the Key Extractor. However, instead of reading just up to 2B from the packet chunk, the extractors of the Context Restoration Unit can read up to 8B each. This is in line with the eBPF VM's registers size (64bit). Furthermore, they replace the lookup key buffer with larger buffers to host either the partially reconstructed stack or the reconstructed registers' state. Finally, instead of an offset, each extractor provides to its downstream extractor the line read from either the Registers Configuration Memory or the Stack Configuration Memory, depending on which of the two parallel pipelines the extractor belongs to. Here, we point out that this approach was needed since the Context Restoration Unit has an additional complexity element when compared to the Key Extractor. The Key Extractor configuration is the same for any received packet, whereas the configuration for the Context Restoration Unit strictly depends on the content of the received packet. Since the entire system is organized in a pipeline, each stage of the Context Restoration Unit has to carry along the configuration to restore the context for the specific packet being processed in that stage.

These two parallel pipelines are fed by a second *splitter* that duplicates the packet chunk. The pipeline that restores the Stack has 10 stages, and it is connected to the Stack Configuration Memory. The line extracted from this memory provides to the Extractors the information needed to populate the stack, including constant values and values extracted from the packet chunk. This information includes: (i) the offset at which the packet chunk should be read; (ii) the operation to be performed on the read byte (and the constant value associated to that); (iii) the target address in the stack. The pipeline that restores the Registers has 10 stages too, including 9 Extractors and a Delay element. This is the case since only 9 registers need restoration ($R1$ - $R9$) and the Warp Optimizer ensures that $R0$ is only read if its value is changed by the loaded XDP program after restoration.[1] The Delay element is required to synchronize the two context restoration pipelines.

5.4 Integration with hXDP

The Warp Engine pipeline ends in hXDP. Here, the Warp Engine waits for hXDP to be available to receive packets. Once that is the case, it first copies the packet data in the hXDP's Active Packet Selector (APS). The packet data transfer is also pipelined, and it happens in synch with the Warp Engine's pipeline. That is, multiple packets' data are moved through the pipeline at each clock cycle, using a 64B datapath (matching Corundum's datapath). Then, we have two possible behaviors. If the current action memory's line contains a forwarding action, then the Warp Engine sets $R0$, and instructs the APS to proceed with packet forwarding. The APS will then carry out forwarding according to the value provided in $R0$. Instead, if the action is a restore context action, then the Warp Engine sets the hXDP's program counter, registers ($R1 - R9$)

[1] Register $R10$ is read-only, and in hXDP it has a constant value. $R0$ is used to store the return value of helper function calls, which are usually the first instruction run by the program after restoration.

Application	*Instructions*		*TCAM*	*Match*	*Max Stack*
	eBPF	**hXDP**	*Entries*	*size [B]*	*size [B]*
L2 ACL	40	27	3	2	6
Router	119	95	9	4	8
Tunnel	283	155	7	4	24
DNAT	228	135	6	6	40
Suricata	138	65	49	12	40
Katran	1398	1013	20	16	80

Table 2: **Tested applications and key metrics**

and stack, and starts the hXDP's Sephirot eBPF Processor. Sephirot will then run the XDP program starting from the instruction pointed by the program counter, and using the provided registers and stack values.

5.5 Implementation

We implemented the Warp Engine design using the latest version of hXDP, which is integrated in Corundum [16], clocked at 250MHz, and targets a Xilinx Alveo U50 FPGA NIC [4]. The Warp Engine is clocked at 250MHz too, and its pipeline is 28 clock cycles long. Since at 250MHz each clock cycle takes 4 nanoseconds, the Warp Engine introduces a fixed 112 nanoseconds of latency to each processed packet. This is a negligible overhead in the vast majority of cases, and it is the only runtime overhead introduced by the Warp Engine.

Our design has several parameters, e.g., the number of Key Extractor stages and the packet chunk size, which may be changed to meet different use case requirements. We summarize them in Appendix, along with the configuration we implemented in this paper, which is driven by the requirements of the 6 use cases we tested during evaluation (cf. Table 2).

6 Evaluation

In this Section we evaluate correctness, optimizations, resources requirements, and performance of our prototype.

6.1 Applications

We use 6 different applications to perform the evaluation, as detailed next. Table 2 summarizes them and reports relevant metrics, including their requirements in terms of Warp Engine's TCAM entries, lookup key size and max Stack size.
L2 ACL (Running example). This is the application we used as running example, and described in Section 3.1.
Dynamic NAT. Network Address Translation (NAT) for flows coming from a LAN and destined to a public network, and reverse translation. The application has two main branches: (i) one for packets originated from the the LAN, and (ii) the other for those coming from the public network.
XDP Router. An implementation of an IPv4/IPv6 router, provided as eBPF application example with the Linux Kernel. It performs parsing of L2 and L3 headers, and then a lookup in two tables to take a packet routing decision.
XDP TX Tunnel. This is another eBPF application example provided by the Linux Kernel. It performs IPinIP encapsulation matching on destination IP address and destination L4 port. A lookup in a hashtable matches on the destination virtual IP address to retrieve the tunnelling information.
Suricata IDS. Suricata [39] is a software Intrusion Detection System (IDS). Among its multiple features, it provides an XDP program that works as a filter, to perform early dropping of undesired flows. The XDP program contains a large number of processing branches to handle all the combinations of stacked 802.1Q and 802.1AD VLAN headers, and performs a lookup in a hashmap to take some of the filtering decisions.
Katran. Katran [14] is an XDP-based Layer 4 load balancer. It encapsulates packets with a specific destination Virtual IP addresses and balances the connections towards the available servers. The first part of the processing includes L3 parsing and handling of ICMP/ICMPv6 protocols. Then, a first map lookup retrieves the virtual IP information. The application uses this information to query a Least Recently Used (LRU) map, in order to fetch the address of a connection table. A query to the connection table finally retries the real IP address of the destination server.

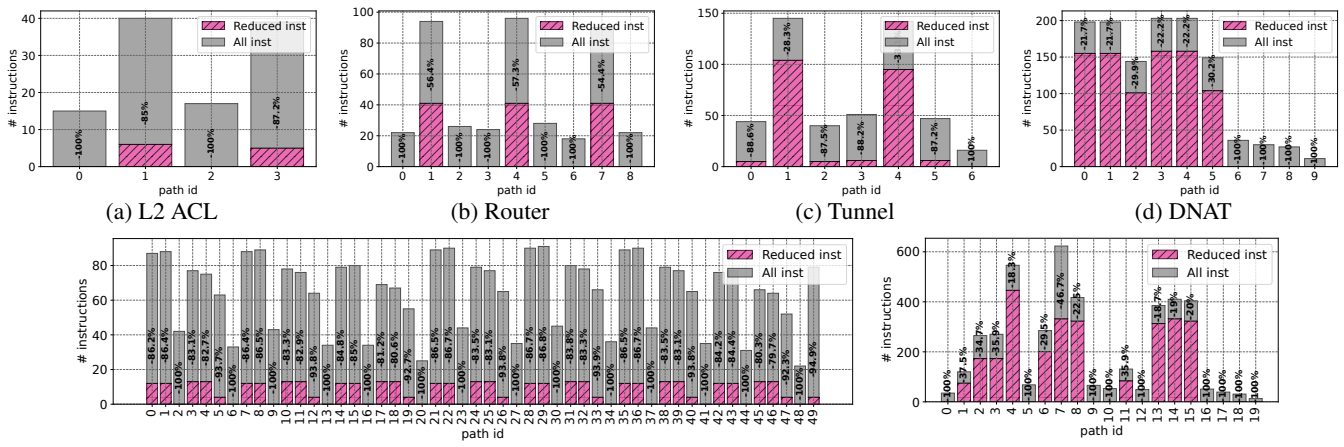
6.2 Functional Equivalence

Program warping modifies a program to run it on a system that comprises two different executors. We therefore performed tests to verify that the resulting behavior matches the original program behavior. In particular, for each of the tested applications, we: (i) enumerate all the program's control paths; (ii) generate input packets that trigger the execution of each of the listed paths; (iii) and finally verify the produced output, for all the generated input packets. We run these steps for the 6 applications described earlier, verifying that program warping keeps functional equivalence. More details are in Appendix.

6.3 Warped instructions

We now evaluate the number of instructions that can be skipped thanks to program warping, since their functionality is implemented by the Warp Engine. This requires evaluating the instructions being actually executed at runtime. We use `uBPF` [22], a userspace eBPF processor, extending it to implement a Warp Engine emulator in software, to compute the number of actually executed instructions for all the tested applications, and for all the control flow paths of each application. Since the control path at later stages of the program depends also on the stored state, e.g., entries in the maps, our testing strategy is adapted to test the multiple possible state conditions. For instance, in the case of the L2 ACL, after the map lookup there are two different paths: if the lookup returns an entry; or not (cf. Listing 1).

Figure 5 reports the results, showing the total number of eBPF instructions executed per path (background bar), and the number of instructions executed when program warping is



(a) L2 ACL (b) Router (c) Tunnel (d) DNAT

(e) Suricata (f) Katran

Figure 5: **Number of instructions per program's control flow path (background bar) vs number of instructions executed per path with program warping (foreground bar). Program warping reduces the instructions to be executed by 50-100%.**

in place (foreground bar). The results show that in many cases the number of instructions to be executed by the eBPF processor is reduced by over 50%, and that anyway in all cases it is reduced by at least 16.3%. More precisely, an application's control flow paths belong to one of two general categories, based on where their execution is going to be implemented: (i) *Mostly Warp Engine*; (ii) and *Mixed*.

Mostly Warp Engine In all the applications there are at least a few control paths whose processing is mostly implemented by the Warp Engine. This is due to the practice of including in the programs several checks to take an early forwarding decision. Many of these early decisions are taken to *protect* programs from bogus or malicious traffic, which has an interesting implication: program warping may provide higher performance boosts when it is the most needed. For instance, some Denial-of-Service attacks' traffic may be entirely handled by the Warp Engine. Consistently, we can observe that all the control flow paths of Suricata fall in this category. In fact, Suricata generates XDP programs to filter network traffic as early as possible. As a result, in some cases the entire program is implemented by the Warp Engine. More generally, in Figure 5e we can see that the instructions reduction depends on the specific path, and it is in the range 82%-100%.

Mixed Some control flow paths split their execution between the Warp Engine and the eBPF processor, either in equal parts or mostly using the latter. This is the case for, e.g., Router, Tunnel, and DNAT. In such cases, the packet parsing and lookup key extraction are delegated to the Warp Engine. The rest of the application logic, e.g., lookup in a single map and packet header mangling, is performed by the eBPF processor. In some programs, this second part is relatively simple. For instance, we see that program warping reduces the instructions of these paths by 54%-57% and 51% for Router (Paths 1, 4, 7). In some applications, this second part is instead more complex. For instance, the Tunnel application's paths 1 and 4 are reduced by 28% and 33%, respectively. This is the case since these paths have a large number of instructions that deal with the packet encapsulation, which happens after a map lookup. Similarly, in the first six DNAT application paths, the reduction ranges between 21%-31% due to the high number of instructions required to recompute the checksum and packet modifications after the program warping. In Katran's paths this is even more evident due to the many lookups in maps performed in the program's paths. The reduction, in this case, ranges from 16.8%-34.7%.

	Logic Res.		*Memory Res.*	
	LUTs	**Reg.**	**BRAM**	**URAM**
Corundum (C) + hXDP	10.7%	6.91 %	13.65%	2.34%
C + hXDP + Warp Engine	16.8%	9.86%	14.51%	8.44%

Table 3: **FPGA resources usage**

6.4 Warp Engine Hardware Requirements

We now evaluate the FPGA resources required by the Warp Engine. We compare the requirements with those of the latest hXDP version [4], which is integrated within the Corundum NIC [16] and targets a Xilinx Alveo U50. The U50 is equipped with a Xilinx Ultrascale+ FPGA, which offers 4 main types of resources that are of interest to us: (i) Lookup-Tables (LUTs); (ii) registers; (iii) block RAM (BRAM); and Ultra RAM (URAM). The LUTs and registers are the main building blocks to implement logic functions, whereas BRAM and URAM are two different memory blocks provided by the FPGA. BRAMs provide multi-port access, whereas URAM have a single read/write port but they are larger than BRAMs. For all the resource types, the Warp Engine is within our original requirement of keeping the packet processing subsystem below the 20% of the available FPGA resources (Table 3).

6.5 End-to-end performance

We finally test the end-to-end system when processing traffic, measuring both packet throughout and forwarding latency.

Testbed We use two machines: a first machine is equipped with a 100Gbps Mellanox ConnectX-5 NIC, and it runs a DPDK-based traffic generator/receiver, capable of sending traffic at 100Gbps with 64B packets, i.e., ∼150Mpps; the second machine is equipped with a single port 100Gbps Xilinx Alveo U50, connected back-to-back with the first machine. In all the tests, we measure packet forwarding that is handled entirely within the NIC, and drops. In this last case, we gain visibility by placing a dedicated drop counter within the FPGA design. Latency is always measured at the packet generator's machine, as difference between the packet reception and packet sent (hw) timestamps. We do not measure the performance of processing that involves transferring packets to the host system, since the Corundum's network driver can forward few Mpps, and it would therefore become the system's bottleneck [16]. However, we remark that in terms of Warp Engine+hXDP design, the transmission to a NIC's port or to the PCIe bus is implemented with the same hardware logic, therefore our system tests are representative of both cases.

Baseline We perform a baseline test to measure throughput and latency when 100% of program's instruction are implemented by the Warp Engine. Since the Warp Engine pipeline performs the same steps for all the applications and execution paths, the performance is the same in all the cases. That is, we achieve ∼83Mpps, when performing `DROP`, and 50Mpps with a 1µs of per packet end-to-end latency when forwarding packets (`TX`). This is the same performance achieved by hXDP when running a program with a single instruction.[2] In fact, the Warp Engine relies on hXDP to carry out the forwarding action, and this performance matches the hXDP baseline performance, confirming that the Warp Engine is never a bottleneck in our design. In fact, this result holds true even when forwarding small packets that are larger than the Warp Engine packet datapath, e.g., 65B long packets.[3]

Applications Wach application has multiple execution paths, which are taken depending on the received traffic and application's state. For paths that are 100% processed by the Warp Engine, the performance is the same of the baseline case, for all applications and paths. This often provides throughput improvements of over 10x for such paths (E.g., see last row of Table 4). For the remaining paths, in the interest of space, we report the performance for only a subset of them, focusing on those that are the most frequent cases, or on cases that are interesting to study the system behavior. In particular, for L2 ACL, DNAT and Katran, we select the paths corresponding to successful lookups in the maps, which are the most frequent cases. For instance, this would be the path taken by established connections in both the DNAT and Katran cases. For Suricata, we see from Figure 5e a periodic pattern, which is due to the repetition of several different packet parsing combinations (e.g., including or not multiple levels of VLAN parsing). Among these, we select the worst case for program warping, i.e., the path corresponding to the most instructions executed by hXDP. Finally, for Router and Tunnel, we analyze traffic traces to select the most common paths that would be triggered by processing such traffic. For Router, we use a Datacenter trace [6], and the path handling IPv4 is triggered in over 80% of the cases. For Tunnel, we use a MAWI trace [28], and the case IPv4+TCP is triggered in 60% of the cases.

We summarize the results in Table 4. For the selected execution paths, program warping improves throughput by 1.23x-3.08x, and increases latency in the worst case by only 104 *nanoseconds*. We can make two important observations. First, program warping provides remarkable throughput improvements, nonetheless, comparing to results from Figure 5, it seems that the tested paths provide a lower-than-expected speed-up. For instance, for the L2 ACL's path #1, Figure 5a shows that only 15% of the instructions should be executed, which would suggest a potential throughput increase of over 6x. However, our test measures a 1.7x increase. This is the case since different hXDP instructions have different costs. For example, a `call` instruction may cost several clock cycles, and it also depends on variables such as the lookup key length. Therefore, the absolute number of instructions at compile time is not necessarily a good estimator for the achievable performance at runtime. Furthermore, it is important to notice that the Warp Optimizer works on the eBPF bytecode, which at a later stage is transformed by the hXDP compiler. The hXDP compiler may remove some instructions and parallelize others, therefore modifying the total program length (cf. Table 2). A side-effect of this is that the *warped* instructions may have been finally removed or parallelized by the hXDP compiler, which reduces the relative gain obtained by avoiding their execution. Second, in the case of Katran we observe a reversed result. Katran's throughput is improved to 2.3x, despite Figure 5f shows only an 18.7% reduction for the path #11's instructions. This is due to the relatively large number of (conditional) jumps in the first part of the Katran's execution path. These jumps introduce bubbles in the hXDP's processor pipeline, lowering throughput and increasing latency. In fact, Table 4 shows that in the case of Katran the Warp Engine significantly improves also forwarding latency, lowering it from 1.9µs to 1.5µs.

To put this in perspective, for the Router, Tunnel and DNAT cases the throughput of hXDP+Warp Engine (clocked at 250MHz) matches that of an Arm Cortex A72's core clocked at 2.75GHz (A processor in use in high-end SmartNICs [32]). For Katran, the throughput is similar to that provided by two A72's cores. In the case of L2_ACL and Suricata, our prototype matches the performance of four A72's cores. More details about this are provided in Appendix.

[2] hXDP takes 3(5) clock cycles to handle drop(forward), with 64B packets.

[3] The interested reader can find more insights about the implications of datapath size on the packet forwarding throughput in [44].

Application [Path ID]	*Latency* [ns] w/ WE	w/o WE	*Tput* [Mpps] w/ WE	w/o WE	*Tput w/WE* vs w/o WE
L2 ACL [#1]	1128	1024	9.26	5.43	170.37%
Router [#7]	1304	1212	3.47	2.66	130.58%
Tunnel [#4]	1368	1288	2.84	2.21	128.41%
DNAT [#2]	1444	1364	2.34	1.89	123.36%
Suricata [#46]	1124	1112	10.86	3.52	308.52%
Katran [#11]	1501	1942	2.08	0.90	231.08%
Performance for 100% instruction reduction scenarios					
Suricata [#23] (DROP)			82.87	4.31	1824,72%

Table 4: **Warp Engine (WE) End-to-End Performance**

7 Discussion

Actual Performance Program warping throughput improvement depends on: (i) the XDP program; and (ii) on which program's control path is executed. In our evaluations, we only measured a subset of the program's paths. In operational settings, other control paths are likely to be part of the workload. In some cases, this may dramatically increase the performance gain. For instance, in Suricata, the last row of Table 4 shows the performance for one of the cases in which the Warp Engine can entirely offload hXDP. In this case, the system provides an 18.2x throughput increase. It is worth noticing that these paths are not necessarily uncommon or rarely executed. On the contrary, often they may represent the most frequently taken paths. For instance, CloudFlare defines XDP programs to perform early packet dropping for DDoS protection [7]. In such applications, these highly boosted paths are expected to be handling the majority of traffic.

Programming for Performance This last observation highlights that the performance of the loaded program is not guaranteed, and instead it depends on the received input. While this is sometimes considered an issue in switching devices [9], this is an expected behavior for software programmers who are already used to handle such variability in performance. A related interesting observation is that programmers can describe processing *rules* using `if` statements and *hardcoded* variables and constants, to improve throughput. This is the same set of techniques used to optimize XDP programs running within the Linux kernel on x86 processors. That is, program warping aligns to both the XDP programming model and best practices to improve program performance, matching XDP programmers' expectations.

Configurability Our current program warping design is quite general, since the 6 presented applications include a good variety of cases. Nonetheless, there may be some other applications for which the Warp Engine resources cannot entirely describe the instructions that can be warped. While falling back to the eBPF executor is always a viable option, we also point out that it is possible to change several parameters of our design to accommodate different applications (e.g., lookup key's size, TCAM entries number, etc.).

Limitations The performance acceleration provided by program warping strictly depends on the share of instructions that a program dedicates to packet parsing and classification. If the majority of the runtime is spent in other parts of a program execution, program warping will only provide small benefits. Conversely, there could be cases in which the warp engine cannot offload all the program instructions that can be in principle *warped*. For example, this is the case if the Key Extractor's pipeline is too short to extract all the data needed for parsing. In such cases, the Key Extractor and Context Restoration Unit's pipelines length provides a hard limit to the maximum number of instructions that can be warped.

Scaling throughput The Warp Engine is not the system's bottleneck. Therefore, for throughput oriented solutions where FPGA resources are available, it is possible to envision a design in which the Warp Engine serves packets to multiple hXDP modules that work in parallel. Similar high-throughput solutions is something we plan to explore as future work.

Portability While in this paper we use program warping to improve hXDP, the approach has more general applicability. In particular, the Warp Optimizer can be decoupled from the underlying hardware platform. For example, the extracted parsing logic can be used to automatically generate packet parsing programs specified with P4, or to map it to DPDK's `rte_flow` API calls, to configure the underlying NIC packet parsing capabilities. Here, a challenge is to describe efficient mechanisms to move the partial *execution context* from e.g., the device subsystem performing header parsing and the subsystem that executes the remaining part of the program. Thus, the benefits of the approach vary depending on the specific target, which opens an interesting opportunity for future research. The Warp Engine design is also portable to different platforms, beyond FPGAs. In fact, it is a parametrized but "fixed" pipeline, thereby requiring relatively little changes to be ported to an ASIC implementation.

8 Related Work

A large number of new NIC designs appeared in the last few years [17, 20, 27, 32, 34, 41]. These solutions mostly combine in a mix-and-match manner different compute and network modules, e.g., regular NIC's switching ASICs with general purpose compute clusters based on RISC cores [32], or FPGA-enhanced switching combined with general purpose clusters [20, 41]. In many of the solutions, a novelty factor is enabling P4-based programming of the switching ASIC. This effectively corresponds to replacing the fixed-function switching module with a programmable switching module [34]. However, in all these designs the data plane needs to be explicitly programmed with the provided tools, e.g., based on P4. Some of these designs offer (partial) eBPF support. However, they implement eBPF on top of the general purpose clusters, replicating the architecture commonly used in server machines, but on a smaller scale (including the need to transfer data from the switching ASIC to the general purpose compute clusters using an internal bus). In research, previous work addresses the challenges of moving data among these modules [25], and explores ways to

leverage these new NIC designs to improve application performance [12, 15, 24, 26, 35, 36, 43]. Program warping focuses specifically on the design of the packet switching module, targeting FPGA NICs, and presenting a solution that integrates with Linux applications that leverage eBPF/XDP. We extend hXDP [10], which to the best of our knowledge is the only solution providing full support for XDP on FPGA NIC directly within the switching module. Compared to hXDP, we provide better performance introducing a new compilation step co-designed with a hardware module, the Warp Engine, which is pipelined to the hXDP processor.

Recent work addressed eBPF programs oprimization at compile time, targeting x86 processors [29, 42]. These works share with us the challenge of performing static analysis of the programs, and leverage some of the insights we discussed about the eBPF execution model. However, they focus on implementing compiler techniques targeting a fixed processor design, whereas we co-design the compiler and the hardware executor. Another related work is Gallium [43], which targets the offloading of a program's part to programmable switching ASICs. Also in this case, it assumes a fixed set of executors, including programmable switching chips and processors. Program warping, instead, introduces both a compiler and hardware design that integrates with the XDP processor, in order to push the intermediate computations context directly within the processor environment.

9 Conclusion

We introduced program warping, a method that leverages compiler-hardware co-design to accelerate the execution of eBPF programs running on FPGA NICs. Program warping enhances existing systems that run eBPF on FPGA NICs with a new compilation step and adding a hardware module, the Warp Engine. The compilation step identifies parts of eBPF programs that can be more efficiently implemented by the Warp Engine, which offloads them from eBPF processors, improving throughput (120%-300%, and up to 18x) at the cost of a small amount of additional FPGA resources. The crucial insight is that only packet data reads and comparisons are needed to implement the identified program parts. Therefore, the Warp Engine supports this minimal set of operations, thereby achieving speed and efficiency, while eventually concluding packet processing on a regular eBPF processor that could handle any more complex program's functions.

Acknowledgements

We thank the anonymous USENIX ATC 2022 shepherd and reviewers for their valuable feedback. This work has been partially funded by the European Commission in the frame of the Horizon 2020 projects 5GMED (grant #951947) and MARSAL (grant #101017171).

References

[1] Cilium website. https://cilium.io.

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

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

[4] Pushing xdp into smartnics. https://fosdem.org/2021/schedule/event/sdn_hxdp_fpga/.

[5] Suricata Documentation. Using Capture Hardware: eBPF and XDP. https://suricata.readthedocs.io/en/latest/capture-hardware/ebpf-xdp.html.

[6] T. Benson. Data set for IMC 2010 data center measurement. http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html.

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

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[9] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 99–110, New York, NY, USA, 2013. Association for Computing Machinery.

[10] 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. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990. USENIX Association, Nov. 2020.

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

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

[13] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall. drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 1–14, New York, NY, USA, 2017. ACM.

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

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

[16] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen. Corundum: An open-source 100-Gbps NIC. In *28th IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2020.

[17] Fungible, Inc. S1 DPU Product Brief. https://www.fungible.com/wp-content/uploads/2021/01/PB0029.01.12020113-Fungible-S1-Data-Processing-Unit.pdf.

[18] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design principles for packet parsers. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '13, page 13–24. IEEE Press, 2013.

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

[20] Intel Corporation. Infrastructure Processing Units (IPUs). https://www.intel.com/content/www/us/en/products/network-io/smartnic.html.

[21] Intel Corporation. 5G Wireless. https://www.intel.com/content/www/us/en/communications/products/programmable/applications/baseband.html, 2020.

[22] IOVisor Project. uBPF repository. https://github.com/iovisor/ubpf.

[23] D. Korolija, T. Roscoe, and G. Alonso. Do OS abstractions make sense on fpgas? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 991–1010. USENIX Association, Nov. 2020.

[24] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 1–14, New York, NY, USA, 2016. Association for Computing Machinery.

[25] 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)*, pages 243–259. USENIX Association, Nov. 2020.

[26] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Marvell Technology, Inc. Data Processing Units. https://www.marvell.com/products/data-processing-units.html.

[28] MAWI. MAWILab traffic trace - samplepoint f - 2021-03-22. https://mawi.wide.ad.jp/mawi/samplepoint-F/2021/202103221400.html.

[29] S. Miano, A. Sanaee, F. Risso, G. Rétvári, and G. Antichi. Dynamic recompilation of software network services with morpheus, 2021.

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

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

[32] NVIDIA Corporation. NVIDIA BlueField data processing unit (DPU). https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[33] Orange. OKO. https://github.com/Orange-OpenSource/oko.

[34] Pensando Systems. Pensando DSC-100 Product Brief. https://pensando.io/wp-content/uploads/2020/03/Pensando-DSC-100-Product-Brief.pdf.

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

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

[37] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM '16*, ACM SIGCOMM '16, pages 15–28. ACM, 2016.

[38] N. Sultana, S. Galea, D. Greaves, M. Wojcik, J. Shipton, R. Clegg, L. Mai, P. Bressana, R. Soulé, R. Mortier, P. Costa, P. Pietzuch, J. Crowcroft, A. W. Moore, and N. Zilberman. Emu: Rapid prototyping of networking services. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 459–471, Santa Clara, CA, July 2017. USENIX Association.

[39] Suricata. Suricata IDS Website. https://suricata.io/.

[40] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*, SOSR '17, page 122–135, New York, NY, USA, 2017. Association for Computing Machinery.

[41] Xilinx, Inc. Alveo SN1000 SmartNIC. https://www.xilinx.com/applications/data-center/network-acceleration/alveo-sn1000.html.

[42] Q. Xu, M. D. Wong, T. Wagle, S. Narayana, and A. Sivaraman. Synthesizing safe and efficient kernel extensions for packet processing. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 50–64, New York, NY, USA, 2021. Association for Computing Machinery.

[43] K. Zhang, D. Zhuo, and A. Krishnamurthy. Gallium: Automated software middlebox offloading to programmable switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 283–295, New York, NY, USA, 2020. Association for Computing Machinery.

[44] N. Zilberman, G. Bracha, and G. Schzukin. Stardust: Divide and conquer in the data center network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 141–160, Boston, MA, Feb. 2019. USENIX Association.

Appendices

A Warp Engine Parameters

In Table 5 we report a subset of the parameters that can be configured in the Warp Engine. This is helpful to accommodate different workloads, or to tune the FPGA resources requirements to the workload of interest in the specific deployment. For instance, there may be cases in which a packet chunk of 64B is sufficient to extract the entire packet context. Likewise, there may be needs to extend the lookup key beyond 16B, etc.

Parameter	Val.	Description
Packet chunk size	128B	Number of packet's bytes that can be read to build the lookup key (also affects the TCAM entries width).
Lookup key size	16B	Size of the lookup key used in the match-action unit
Key Extractor Stages	12	Corresponds to the maximum number of different places that can be read in the packet chunk
Key Extractor read size	2B	Maximum number of contiguous bytes that can be read by each Key Extractor's stage (also affects the maximum size of the constant value used for the bitwise operation)
TCAM entries	64	Maximum number of match-action entries that can be configured by the Warp Optimizer for a given program
Stack Extractor Stages	10	Corresponds to the maximum number of different places that can be read in the packet chunk
Stack buffer	136B	Maximum number of stack bytes that can be restored
Stack Extractor read size	8B	Maximum number of contiguous bytes that can be read by each Stack Extractor's stage (also affects the maximum size of the constant value used for the bitwise operation)
Reg. Extractor Stages	9	Corresponds to the maximum number of different places that can be read in the packet chunk
Reg. Extractor read size	8B	Maximum number of contiguous bytes that can be read by each Stack Extractor's stage (also affects the maximum size of the constant value used for the bitwise operation)

Table 5: Warp Engine's main design parameters, and the values used for the design tested in this paper.

B Applications

Here we report a slightly more detailed description of the application used during our system evaluation (and reported in the paper).

L2 ACL (Running example). This is the application we used as running example, and described in Section 3.1. It includes three branches: the main processing branch handles IPv4 packets and checks whether the source MAC address is present in the access list; the other two branches handle IPv6 packets, which are always dropped), and any packet that is not IP, which is passed to the networking stack.

Dynamic NAT. Network Address Translation (NAT) for flows coming from a LAN and destined to a public network, and reverse translation. The application has two main branches: (i) one for packets originated from the the LAN, and (ii) the other for those coming from the public network. When a flow's first packet from the LAN is processed, the application selects a new *NATed* port, and saves it in the NAT binding table using the 5-tuple as flow identifier. Then it performs address translation and forwards the packet. For any following flow's packet, the application retrieves the NATed port, and performs address translation accordingly. In a similar way, packets from the public network are subject to a reverse NAT if there is a corresponding entry in the NAT binding table, or they are dropped otherwise.

XDP Router. An implementation of an IPv4/IPv6 router, provided as eBPF application example with the Linux Kernel. It performs parsing of L2 and L3 headers, and then a lookup in two tables to take a packet routing decision. The first table is an exact match table that looks up the entire IP destination address. If the lookup in the first table fails, the application performs a second lookup in a Longest Prefix Match (LPM) table.

XDP TX Tunnel. This is another eBPF application example provided by the Linux Kernel. It performs IPinIP encapsulation matching on destination IP address and destination L4 port. The application works with both IPv4 and IPv6, with the two main processing branches handling these two cases to assign the proper IPv4 or IPv6 encapsulation. A lookup in a hashtable matches on the destination virtual IP address to retrieve the tunnelling information.

Suricata IDS. Suricata [39] is a software Intrusion Detection System (IDS). Among its multiple features, it provides an XDP program that works as a filter, to perform early dropping of undesired flows. The XDP program contains a large number of processing branches to handle all the combinations of stacked 802.1Q and 802.1AD VLAN headers. After VLAN parsing, it processes differently IPv4 and IPv6 packets, and performs a lookup in a hashmap providing the 5-tuple plus the (optional) VLAN identifiers to take some of the filtering decisions.

Katran. Katran [14] is an XDP-based Layer 4 load balancer. It encapsulates packets with a specific destination Virtual IP addresses and balances the connections towards the available servers. The first part of the processing includes L3 parsing and handling of ICMP/ICMPv6 protocols, for early response to echo request messages. Then, a first map lookup retrieves the virtual IP information. The application uses this information to query a Least Recently Used (LRU) map, in order to fetch the address of a connection table. A query to the connection table finally retries the real IP address of the destination server. If a destination is not found, and the packet has the SYN flag set, then Katran installs a new forwarding rule in

the connection table to ensure forwarding consistency for the following packets of that flow.

C Software Emulator

For some of the Warp Optimizer evaluations, we used a Warp Engine emulator based on `uBPF` [22]. Here we give additional details about such implementation.

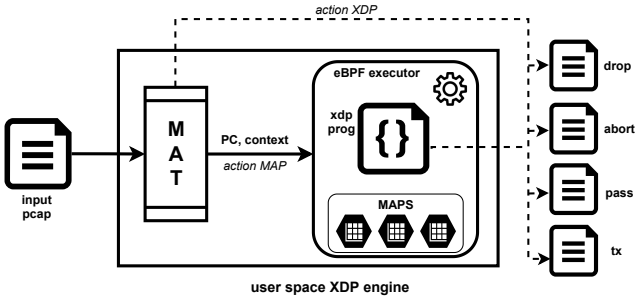


Figure 6: MAT+uBPF Architecture

`uBPF` is an open source project that implements an eBPF processor in userspace. Unfortunately, out of the box, `uBPF` misses relevant functional blocks, such as maps. We therefore used OKO [33], an open source project providing an eBPF engine for OpenVSwitch, to enhance `uBPF` with the OKO's maps and helper functions implementations. Starting from this basis, we further extended this implementation to include any additional feature required by the Warp Engine (cf Figure 6).

Our implementation takes several input files to configure its internal modules. First, we implemented a program that reads the ELF files provided by the standard LLVM eBPF compiler, and which creates as output:

1. a text file containing the *eBPF instructions*, formatted as a sequence of bytes representing the 64bit instructions of the program;
2. a JSON file describing the XDP program's map definitions. Such information includes the key size, value size, number of entries and type of map (array, hashmap, etc.).

These two files are provided to our software emulator to configure the eBPF executor, and create the maps required by the program.

The emulator's Warp Engine module implements a Match Action Table (MAT) with ternary match values. The MAT is configured using the output generated by the Warp Optimizer.

uBPF execution The emulator takes packet in input by reading a PCAP file. It can then be run in two different modes of operation: with the software Warp Engine disabled; and with the software Warp Engine enabled.

In the first case, the MAT is bypassed and the packet is directly fed to the eBPF executor, which applies the eBPF instructions on the packet data. With the MAT enabled, instead, the packet data is used to extract the fields needed to perform a lookup in the MAT. The matched entry contains the action that must be performed on the packet, that is:

- a standard XDP return code (DROP, ABORT, PASS and TX), so in this case the packet bypasses the eBPF execution stage;
- a context restoration action, which contains the information to construct and restore the context.

In case of context restoration, we copy the registers and stack values as described in the action (which is configured from the Warp Engine's output). Then, in any case, the program counter is updated with the one required by the matched action, and the regular eBPF execution starts. Finally, the emulator outputs a number of global and per-packet statistics:

- one PCAP trace for each XDP return code, for packets that have been subject to the DROP, PASS, ABORT and TX XDP actions;
- the list of instructions executed for each packet;
- the number of instructions actually executed;
- the number of times a rule in the Match Action Table has been matched.

These statistics have been collected to construct the results shown in Section 6.

D Functional Equivalence

We provide more details about the strategy we used to check functional equivalence of programs running with and without program working. In particular, we validate the equivalence of running an eBPF program in our accelerated system and running the same program in the standard Linux kernel XDP implementation. First, we analyze the *behavioral equivalence*, i.e. that the packets out of the Linux kernel implementation exactly match the packets in output from our software implementation. This is a black-box test and has two outcomes: (i) it validates the equivalence between the standard implementation and our accelerated version, and (ii) it validates the correctness of our software prototype. For what concerns the test cases, for each application we use synthetic packet traces in which each packet exactly matches one entry in the Match Action Table. We run an eBPF program with and without the MAT enabled and compare the output packet traces. We obtain the behavioral equivalence by verifying that the two outputs match exactly, in terms of packet data and associated XDP action.

Nonetheless, a careful choice of the test cases should take into account all the possible inputs such that the totality of the eBPF instructions of a program are covered. For example, in the NAT application, the first packet of a new connection matches the same entry of subsequent packets, but for the first packet the processing is different, and the instructions covered are different as well. In other words, we should take into account the state updates in the execution of a program. To validate the correctness of our test cases to cover the entire set of program's instructions, for each application we crafted

the packets to cover all the branches in the Control Flow Graph, along with the correct configuration of the eBPF map entries. For every use case considered, we achieved the full program instructions coverage. We verified this by checking that the enumerated instructions represent the totality of the original program.

In the case of Katran, we used a simplified version by removing some parts of the code for which the instructions could not be executed. For example, some portions of the Katran code rely of timeouts triggering a certain condition. Since our uBPF prototype does not implement timers, we removed those program parts. In any case, all the instructions that are not covered by our tests always happen after the *warped* part of the program, therefore we could still verify that program warping does not modify in any way a program's behavior.

E Comparison with commercial SmartNICs

In Section 6 we compare our prototype only with hXDP. This is the case since we are not aware of any other NIC platform that supports running unmodified eBPF in the network data plane.[4] Furthermore, we are interested in evaluating the specific contribution of program warping to FPGA-based eBPF executors, and less concerned with the evaluation of packet processing when using different platforms. However, when looking more generally supporting eBPF on a NIC, some recent commercial NICs that include battery of general purpose CPUs can indeed run unmodified XDP programs.

It should be clear that comparing our program warping prototype with such systems is not generally correct from a technical perspective, since the goals, constraints and scopes of application are too different to devise a fair testing strategy. For instance, a more direct comparison of the packet forwarding data plane component would require an ASIC-based implementation of program warping.

While we are aware of the above, we believe that the comparison may still be useful for practitioners who may be interested in evaluating FPGA NIC solutions vs alternatives. Therefore we decided to at least include such tests in this appendix for the interested reader.

NVIDIA Bluefield2 architecture We tested the XDP programs performance on an NVIDIA Bluefield2 NIC [32] (in NVIDIA terminology, these devices are currently called Data Processing Units, or DPUs). The Bluefield2 combines two main subsystems: a switching data plane based on the Mellanox ConnectX6 architecture; and a battery of 8 general purpose Arm A72 CPUs running at up to 2.75GHz. The ConnectX6 receives the packets from the network ports and can forward them directly to the host system, like a regular NIC, or it can re-direct them to the Arm CPUs. Here, further processing can happen, and the packets can be either *consumed* locally, or sent back once more to the ConnectX6, to be finally delivered to the host system or to the network port.

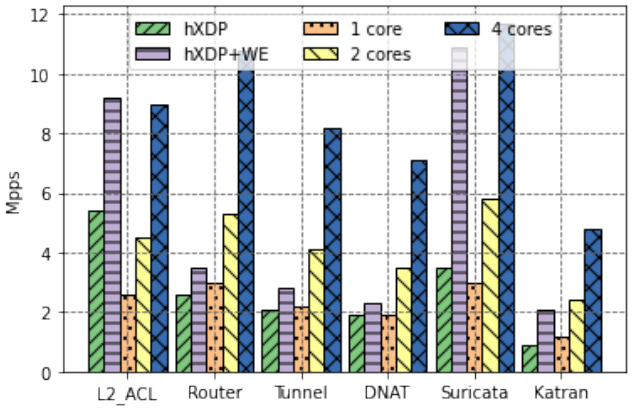


Figure 7: **Forwarding throughput for the applications described in Section 6, when running on hXDP, hXDP+Warp Engine, and on 1-4 cores of an NVIDIA Bluefield2's CPUs.**

Experiments We use the experimental setup, applications and testing strategy described in Section 6 to evaluate the packet forwarding performance of the Bluefield2, when using the Arm CPUs. Figure 7 shows the results.

Like already explained in Section 6, we report the results only for a subset of the application paths. For Router, Tunnel and DNAT, the hXDP+Warp Engine combination achieves a higher throughput than an Arm core clocked at over 10x the hXDP clock frequency. For Katran, our prototype is close to the performance provided by two cores. Finally, in the case of L2_ACL and Suricata, the hXDP+Warp Engine achieves a forwarding throughput roughly equivalent (or close) to 4 Arm cores instead.

These results show that in favorable cases program warping can indeed boost the performance of an FPGA-based processor to match that of several hardcoded cores running at much higher frequency. This suggests that a practitioner will require a careful workload analysis if the choice of an FPGA NIC is not mandated by other deployment requirements[5], since the performance is use case dependent. In any case, program warping provides a viable solutions to run eBPF software packet processing in environments where an FPGA NIC is required.

[4] In fact, Netronome SmartNICs [31] support eBPF, but only in a limited form, and therefore packet processing programs need to be rewritten for the specific Netronome's capabilities.

[5] Requirements may not be necessarily related to the need of hosting FPGA-based accelerators. They may also include considerations on power consumption and type of board cooling.