Accelerating Network Analytics with an on-NIC Streaming Engine

Sebastiano Miano[a,⋆], Giuseppe Lettieri[b], Gianni Antichi[a], Gregorio Procissi[b]

[a]Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Milano, Italy
[b]Università di Pisa, Dipartimento di Ingegneria dell'Informazione, Pisa, Italy

Abstract

Data Stream Processing engines have recently emerged as powerful tools for simplifying the analysis of network telemetry data. Motivated by the ever-growing volume of data requiring analysis, cutting-edge approaches integrate them with programmable switches to filter out less relevant traffic and enhance their processing capabilities.

In this paper, we propose an alternative solution: leveraging SmartNICs as high-performance accelerators for stream processing operations. SmartNICs are commonly deployed in datacenter networks, and their architecture is often characterized by numerous low-power processors that align seamlessly with the highly parallelizable computational requirements of standard streaming analysis frameworks.

Starting from WindFlow, a state-of-the-art stream processor, we present an innovative architecture that enables the offloading of a portion of its computation to a commodity Netronome SmartNIC. We implemented the offload logic using eBPF, making our solution compatible with any NIC supporting this programming paradigm. We developed a diverse range of applications (i.e., flow metering, port scan detection and SYN flood attack detection) and show that our solution can analyze up to 40% more traffic compared to a pure software approach.

Keywords:
Stream Processing, Computation Offload, SmartNICs, Accelerated Data Path, eBPF/XDP

1. Introduction

Advancements in data plane programmability have allowed operators to gather fine-grained telemetry data from each switch to be then consolidated at logically centralized collectors to gain network-wide view [37, 29, 22]. As datacenter networks can encompass hundreds of thousands of switches [17], with each switch producing millions of reports per second [55], the sheer volume of data for analysis has become a major bottleneck [26, 48, 55]. As a consequence, dealing with such an ever-increasing data volume presents challenges in scaling data collection and analysis for telemetry systems [26, 48, 55].

Recent developments have demonstrated the effectiveness of combining Data Stream Processing (DaSP) frameworks with programmable switches to capture and analyze telemetry traffic at scale [19]. DaSP frameworks offer a simple programming abstraction well-suited for standard traffic analysis tasks, as demonstrated in the context of heavy hitter detection, super-spreader detection, and port scanning [19], among others. Programmable switches contribute by efficiently filtering out less relevant traffic that could otherwise overwhelm the host running the DaSP framework. Moreover, it's worth noting that most DaSP frameworks are built upon the Java Virtual Machine (JVM) to facilitate the implementation of streaming applications on distributed environments, specifically homogeneous clusters. This approach introduces notable overheads, such as serialization/de-serialization and cluster management. These overheads may not be easily offset in the context of network data analysis, where the input rate of reports is continuous, and the computational workload is moderate.

⋆This work was partially supported by the Italian Ministry of Education and Research (MUR) through the ForeLab project (Departments of Excellence) and the PRIN project NEWTON (Project no. 2022ZA8T22), by the University of Pisa under the "PRA – Progetti di Ricerca di Ateneo" (Institutional Research Grants) – Project no. PRA_2022-2023_91 INTERCONNECT, and by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART").

Email addresses: sebastiano.miano@polimi.it (Sebastiano Miano), giuseppe.lettieri@unipi.it (Giuseppe Lettieri), gianni.antichi@polimi.it (Gianni Antichi), gregorio.procissi@unipi.it (Gregorio Procissi)

In this paper we propose a different approach: instead of leveraging programmable switches to scale-out processing, we ask whether it is possible to directly offload part of the DaSP analysis logic into commodity SmartNICs. Our idea is motivated by two main observations: (1) SmartNICs are now commodity in datacetenter networks [16], while the adoption of programmable switches remains limited, and Intel has recently discontinued the Tofino product line [46]; (2) conventional System-on-Chip (SoC) NICs typically feature numerous low-frequency processors (e.g., the Netronome NFP-4000 boasts 60 cores with up to 8 threads each [39]), aligning well with the processing paradigm employed by standard DaSP frameworks, which heavily rely on parallel computation.

To investigate this approach, we selected WindFlow [31], a stream processing library designed for multi-core systems based on the C++17 standard, as a representative DaSP framework, and we examine the benefits of offloading portions of its computation pipeline to a Netronome SmartNIC [39]. Our choice is motivated by the fact that in the context of traffic analysis it has been shown that WindFlow is the state-of-the-art solution [12], outperforming popular competing solutions like Spark [3], Storm [1] and Flink [2]. To maximize the applicability of our approach, we implemented the offload using eBPF [23], the de-facto programming language for end-host networking supporting different use-cases [32, 41, 34, 36] as well as user-space, in-kernel and NIC programming. As a consequence, any NIC with eBPF support can adopt our solution.

We implemented a number of traffic analysis use-cases: (1) flow metering; (2) port scan detection; and (3) SYN flood attack detection. We show that our approach can ingest and analyze up to 40% more traffic compared to a pure software approach.

2. Data Stream Processing and WindFlow

Stream processing is a computational paradigm designed for handling continuous streams of data in (near) real-time. This approach involves creating applications composed of interconnected processing functions known as *operators* or *transformations*, forming a Directed Acyclic Graph (DAG) that defines the entire computational process, as shown in Figure 1.

In this graph, nodes represent stateless or stateful operators responsible for processing data elements, often referred to as *tuples*. These operators receive input from one or multiple streams, transform the data, and produce one or more output streams. These data transfers between operators are represented as arcs, and are

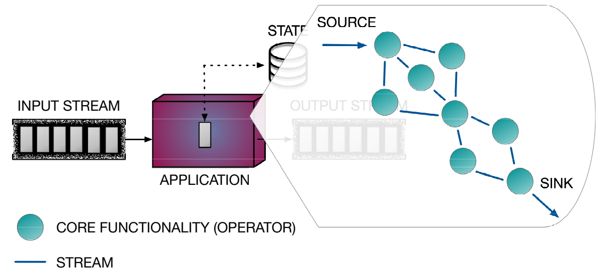

Figure 1: Data-flow graph representation of a stream processing application.

typically implemented using First-In First-Out (FIFO) queues, facilitating data transfer between operators. By executing processing elements in parallel over different stream elements, the execution of streaming applications is expedited.

In our research, we employ WindFlow, a stream processing library explicitly designed for single machines equipped with multi-core CPUs. WindFlow offers several features that enable seamless scaling across multiple processing cores, resulting in notable improvements in both performance and expressiveness.

Indeed, WindFlow provides a diverse range of operators that can be connected to form the data-flow graph of an application. These operators include common streaming functions like *map* and *filter*, as well as advanced parallel operators optimized for sliding-window computations. These operators can be internally replicated to increase their throughput, with each replica working on a subset of the inputs received from the previous operator. Tuples, which are implemented as key-value pair records, serve as the data elements within the graph.

Table 1 reports a subset of the operators offered by WindFlow[1]. The graph must always begin with one or more `Source` nodes, in which tuples are created, and terminates with one or more `Sink` nodes, where tuples are disposed and the associated memory is recycled. The `Filter` node filters all the tuples not respecting a user-defined predicate. The `Map` produces one output per input while the `FlatMap` produces zero, one or more outputs per inputs (inputs and outputs may have different data types). In addition to those basic operators, some applications require to periodically repeat user-defined computations over finite portions of the stream, having the form of *moving windows*. WindFlow includes specialized operators like `Keyed` and `Parallel Window` for expressing window-based computations and

[1]The full list of WindFlow operators is available at this link: `https://paragroup.github.io/WindFlow/operators.html`.

Source

Source	Generates a sequence of streaming items of the same type.

Basic Operators

Map	Applies a one-to-one transformation, producing one output for each input.
Filter	Applies a transformation producing zero or one output for each input.
FlatMap	Produces one or more outputs for each input.

Window-based Operators

Keyed Window	Executes window-based functions. Windows of different keys are processed in parallel, while windows of the same key are sequential.
Parallel Window	Executes window-based functions. All windows, regardless of key, are processed in parallel.

Sink

Sink	Absorbs the input stream of items, all of the same type.

Table 1: Subset of the standard operators available in WindFlow.

to executing them in parallel when windows activate very frequently.

Applications in WindFlow are developed using the `MultiPipe` construct, which establishes multiple parallel pipelines containing operator replicas. These replicas can communicate with either a single replica or all replicas of the subsequent operator. When communication involves all replicas of the next operator, tuple distribution follows a `key-by` approach, where tuples with the same key are forwarded to the same replica of the receiving operator, facilitating consistent data handling.

WindFlow offers two methods for inserting operators into the graph: the default approach adds (using the `add` method) the operator at the end of the `MultiPipe` graph, running it on a dedicated thread and communicating with the previous node through Single-Producer-Single-Consumer (SPSC) lock-less queues. Alternatively, for optimizing performance with more threads than available cores, WindFlow provides the "chaining" mechanism, which merges replicas of different operators into the same thread, optimizing core usage.

By default, SPSC queues in WindFlow have a

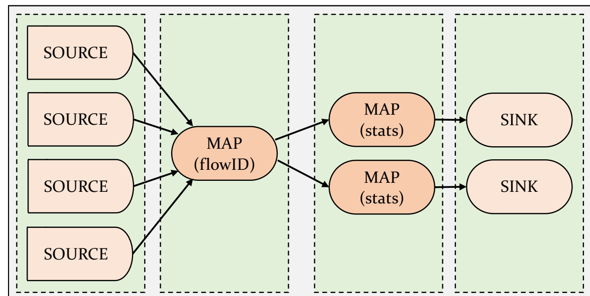

Figure 2: A WindFlow application for Network Analytics.

bounded capacity and operate in a non-blocking mode, often involving busy-waiting. Optionally, WindFlow supports a *batching* mode, where a configurable number of tuples is transferred in a single operation, providing additional flexibility in handling data transfers.

2.1. WindFlow for Network Analytics

Figure 2 shows how WindFlow operators are instantiated to implement a Flow Meter network analytic application ([13]). The purpose of the application is to collect basic per-flow statistics such as number of packets and number of bytes received. The source operators capture packets from one or more network interfaces, parse them to extract the relevant fields (protocol, source and destination address, source and destination port, packet length, and so on) and generate the tuples that are sent to the rest of the pipeline. A Map operator then computes a flow identifier (FlowID) from a subset of the fields and appends it to each tuple. Other Map operators further down the pipeline internally store a hash table that maps FlowIds to their set of statistics. They use the FlowID to look up and update the statistics, creating new entries when they detect a new flow. Finally, the Sink operators terminate the pipeline by recycling the tuple memory.

The general setup of Figure 2 can be found in other network analytic applications, with small variations. Some applications are only interested in a subset of the incoming packets. For example, a Port Scan Detection application is only interested in TCP packets with `SYN` or `SYN-ACK` flags active. In these cases, a barrier of Filter operators follows the sources to select only the matching tuples and discard all others, thus reducing the load on the rest of the pipeline. Other applications may use Window-based Operators instead of the basic Maps used in Figure 2. Section 5 describes three representative applications in which these differences are at work. However, the FlowId operator is typically implemented by all of them. This is because the FlowID can be used as a key to partition the stream of tuples,

creating parallel paths in the pipeline immediately following the FlowID operator. This is possible whenever the processing of one flow is independent of the other flows, allowing the more CPU-intensive computations to be spread on the available cores.

3. NIC-Accelerated Streaming Analytics

Smart Network Interface Cards (SmartNICs) represent a specialized class of devices that couples traditional network connectivity with programmable processing capabilities. Unlike standard NICs, which primarily handle basic packet processing tasks, SmartNICs offload and accelerate complex network-related operations such as packet filtering, load balancing, encryption/decryption, and more, directly within the card itself. The integration of programmable capabilities empowers SmartNICs to substantially enhance network performance, alleviate CPU overhead, and facilitate a wide range of networking and security functions. Consequently, they find optimal utility in data centers, cloud computing environments, and other demanding high-performance networking contexts.

Our primary objective here is to develop a versatile offload methodology that remains agnostic to specific hardware dependencies and circumvents the need for proprietary mechanisms to communicate the offloaded results back to the end host. Both of these requirements naturally find their solution through the utilization of extended Berkeley Packet Filters (eBPF) [23] and the eXpress Data Path (XDP) [20].

In essence, eBPF programs function as software modules executed whenever the software reaches defined software hooks. Within the networking domain, these software hooks is mainly represented by the eXpress Data Path (XDP). In the *native mode*, eBPF programs operate within the kernel space of Linux hosts. Conversely, in the *offloaded mode*, these programs can be executed directly on compatible hardware, such as SmartNICs.

3.1. Netronome Agilio CX 2X40Gbps SmartNIC

One cost-effective and relatively popular SmartNIC that offers eBPF support is the Netronome Agilio CX 2x40Gbps SmartNIC. The high-level architecture of this SmartNIC is depicted in Figure 3.

At its core, this NIC is built around the NFP-4000 processor, featuring 60 Flow Processor Cores (FPCs) with up to 8 cooperatively multithreaded threads per core. The original eBPF hardware offload support was introduced in kernel version 4.9 inside the Netronome

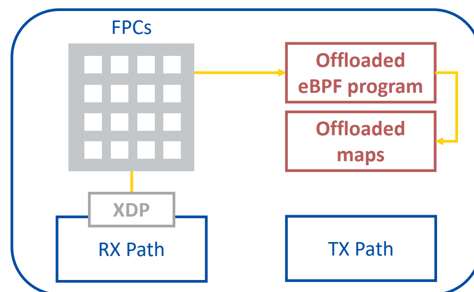

Figure 3: The Netronome Agilio CX SmartNIC design with eBPF hardware offload.

Flow Processor (NFP) driver, and subsequent driver versions (after v4.16) also offer map offload support. The upstreamed kernel driver facilitates the translation of the kernel eBPF program into microcode, which can then be transferred onto the SmartNIC using the NFP eBPF Just-in-Time (JIT) compiler. Consequently, users can effortlessly offload programs without necessitating an in-depth understanding of microcode or the intricacies of the NFP architecture. This streamlined eBPF-based approach empowers users to harness the full capabilities of the SmartNIC with ease.

Inside the SmartNIC, eBPF programs are typically executed on 50 of the overall 60 cores, with each core running 4 threads. These flow processing cores employ a RISC instruction set that is optimized for networking, which closely resembles the eBPF bytecode. This congruence ensures the feasibility of offloading [38]. However, it is essential to acknowledge that the single-thread performance of these cores is notably lower than that of the host.

Lastly, the device is equipped with high-performance PCIe interfaces that facilitate high-speed data and packet transfer between the host system and the NFP.

3.2. Challenges

3.2.1. Resource constraints

One of the primary challenges stems from the inherent resource constraints of SmartNICs in comparison to their host counterparts. The SmartNIC environment imposes stricter limitations, necessitating meticulous attention to several stringent constraints during the offloading process.

Program Size Restrictions The Netronome SmartNIC imposes strict limitations on the size of XDP programs and eBPF maps that can be effectively offloaded. This is primarily due to the significantly limited memory resources available on SmartNICs compared to host systems. Presently, each eBPF program offloaded onto the

SmartNIC can accommodate a maximum of approximately 8,000 instructions [27]. In contrast, eBPF/XDP programs running on the host (software XDP) can support up to 1 million instructions.

Tail Call Mechanism To address program size limitations, eBPF provides a mechanism known as *tail call*, which enables transitions from one program to another within the eBPF context, with each program being independently verified. However, the Netronome SmartNIC presently restricts the offloading of only one XDP program [27], significantly limiting the effectiveness of this optimization.

eBPF Map Size Constraints Additionally, the size of eBPF maps faces limitations within the hardware environment. Specifically, the hardware enforces a maximum limit of 64 bytes per entry, encompassing both the key and value components. Furthermore, regardless of the size of individual entries and the number of maps, an overarching constraint is imposed on the total number of entries in the maps, capping it at approximately 3 million entries.

In the context of streaming algorithms, these limitations demand careful consideration. As discussed in Section 2, WindFlow orchestrates a graph of operators chained together to form the final application. Offloading all operators to the SmartNIC can result in a complex program being deployed, quickly reaching the maximum allowable number of instructions. Moreover, streaming algorithms are designed for long-term execution, often involving metrics collection and advanced calculations. Executing such operations entirely on the SmartNIC may require more memory than is available on the NIC or exceed the limit of maximum key/value entries in the maps, especially with complex protocols such as IPv6.

As detailed in Section 4, we address this challenge by adopting a *partial offloading model*. In this approach, only a portion of the computation, specifically tasks that involve repetition and computationally intensive processes like packet parsing and hash calculation, are offloaded onto the SmartNIC. The SmartNIC then provides packets and additional metadata to the userspace pipeline, where the remaining processing and data collection occur.

3.2.2. Feature constraints

In WindFlow, tuples traversing the processing graph typically have timestamps attached. Depending on the adopted notion of time, the timestamp attribute can either be assigned by the framework or extracted from the input itself (*event time*).

When offloading these operators to a SmartNIC, it can be advantageous to leverage the timestamp attached by the NIC to enhance performance. In the context of eBPF, the `bpf_ktime_get_ns` *helper function*[2] can be used to retrieve the time elapsed since system startup in nanoseconds. This timestamp can then be attached to the packet and used as ingress time in the WindFlow pipeline. However, existing SmartNICs with eBPF offload support have limited features that are currently supported. This limitation encompasses instructions, helper functions, and eBPF map types.

For instance, the Netronome SmartNIC does not support per-core eBPF maps, which makes the process of storing and retrieving eBPF maps information from userspace not as efficient as using userspace data structures. Moreover, it does not support the `bpf_ktime_get_ns` helper function, necessitating our offloading mechanism to handle this case by providing a software timestamp when it is not supported by the hardware.

It is essential to note that recent kernel versions (v6.3+) have introduced functionality to allow XDP to access additional metadata provided by the specific driver [9]. As explained in Section 4, our proposed offloading mechanism addresses this by exploiting hardware-provided metadata when available.

3.2.3. Handling window-based operators

As mentioned in Section 2, among the various operators, WindFlow provides *window-based* operators, particularly useful for applications requiring periodic user-defined computations over finite portions of the stream (e.g., port scans [24], Slowloris Attacks [51]).

When offloading such operators to eBPF, triggering specific computations at defined times can be challenging. The eBPF engine follows an event-based execution model, where packet receipt in the RX path of the NIC triggers program execution. While recent kernels (5.15+) support `bpf_timers` [47], enabling the attachment of timers to eBPF maps that can be triggered at specific time intervals, this approach may not be viable for completely offloading window-based operators for at least two reasons: *(i)* none of the SmartNICs with eBPF offload currently support `bpf_timers`, and *(ii)*

[2] The eBPF subsystem employs a varying number of *helper functions* to interact with other parts of the kernel subsystem. These helper functions are a crucial part of the eBPF APIs, which are integral to the security sandbox provided by the eBPF verifier. The verifier checks the safety of eBPF programs, ensuring they can be safely injected into the kernel. Therefore, calls to external functions are only permitted if they are part of the eBPF APIs.

window-based operators within WindFlow may require more complex data handling than what can be achieved with timers attached to specific maps.

As detailed in Section 4, our *partial offloading architecture* keeps time-based analysis in userspace within the WindFlow pipeline. It offloads only a part of the computation that involves repetitive and computationally intensive tasks, such as packet parsing and hash calculation, to the eBPF hardware program. This approach reduces overall computational overhead in the userspace application and enhances overall performance.

4. Offloading WindFlow graph to Netronome

In this section, we present the architecture and details of our approach for offloading a subset of the WindFlow stream processing graph to Netronome SmartNICs, enhancing the efficiency of stream data processing.

Figure 4 provides an overview of the proposed offload mechanism.

When a packet arrives inside the SmartNIC, it triggers the execution of an offloaded eBPF program, as depicted on the left side of Figure 4. This program implements the offloaded operators, which form a subset of the operators used in the userspace WindFlow pipeline[3]. It then extracts necessary information and generates *tuples* for each received packet, along with hardware-specific metadata (e.g., hardware timestamps or flow IDs) [9] for use within the application's graph.

After the eBPF program's execution, the *tuples* are redirected to specific RX queues, selected by setting the `rx_queue_index` on the hardware XDP context object [40]. These queues are managed using the offloaded eBPF `indirect_map`, which is programmed using the userspace `BPF` APIs.

On the userspace side, `nethuns` [4] sockets are allocated to the hardware queue of the physical network device, indexed based on each replica's corresponding index. The `nethuns` library provides a unified programming abstraction that simplifies access and management of network operations across various I/O frameworks, including `AF_XDP`, `netmap`, and `Libpcap`.

In the case of `AF_XDP`, an additional XDP program is loaded on the end-host kernel (not shown in Figure 4) to redirect received *tuples* to the configured `AF_XDP` socket, seamlessly managed by `nethuns`.

[3] Although Figure 4 shows the offloaded operators as separate entities in the eBPF engine of the SmartNIC, they are implemented as a single XDP program, since Netronome does not support a cascade of eBPF programs to be offloaded into the SmartNIC, as described in Section 3.2.

Finally, the Sniffer object captures received *tuples* from `nethuns` sockets and initiates the WindFlow pipeline on the userspace side, where the remaining operators are executed using information extracted in the hardware pipeline.

4.1. Overview of offloaded operators

Source offload. The starting point for offloading is the Source operator, where an offloaded eBPF program parses the header of each incoming packet and constructs the required tuple for the monitoring application. This enables parallel execution across the numerous cores available on the Netronome card. The constructed tuples are then forwarded to the user-space pipeline for further processing.

The communication between the eBPF program and the userland can occur in two ways: *(i)* through the *control plane*, using eBPF maps [8]; *(ii)* over the data path, forwarding the newly crafted tuple as regular packets. We have chosen the second option for both performance and generality reasons. Firstly, communication via eBPF maps is only efficient on a per-core basis, which is not suitable for our setup since the Netronome card does not support per-core maps. In fact, the large number of cores in the SmartNIC would likely exceed the number of cores on the host PC, leading to significant synchronization issues. Secondly, employing the regular data path logically separates the offloaded processing from the main computation graph.

The Source node in the user-space graph still initiates the WindFlow pipeline, but now functions as an adapter, receiving pre-formed tuples for delivery to subsequent stages after proper type casting.

FlowID offload. To extend the offloading mechanism, we also incorporated the computation of the FlowID within the eBPF program, adding the result as an additional field in the packet-tuple. This approach maintains core affinity with user-space graph computations. Many WindFlow operators indeed use the `key-by` approach to send all inputs having the same key attribute (e.g., a specific field of the tuple) to the same replica. By computing the FlowID within the eBPF program before the tuple enters the user-space pipeline, we then establish a direct connection between the Source and the subsequent operators, resulting in significant benefits in terms of operation synchronization and cache locality.

Please note that, as shown in Figure 4, even when we offload the Source and the FlowID to the SmartNIC, the WindFlow pipeline still needs to be initiated in the Source node. However, these nodes now function as mere adapters that receive already formed tuples to be delivered to the next stages after proper type casting.

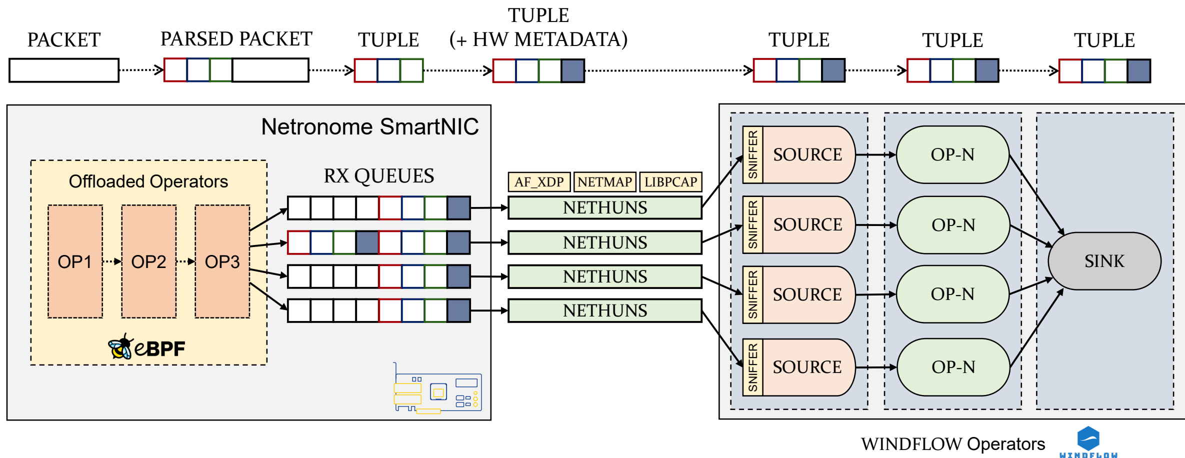

Figure 4: Architecture of proposed WindFlow offloaded mechanism. The SmartNIC hardware eBPF engine executes a subset of the WindFlow operators, extracing *tuples* and hardware metadata from received packets. After the execution on the hardware pipeline, *tuple* are forwarded to the WindFlow userspace graph using the `nethuns` [4] library, which supports different I/O framework such as `AF_XDP`, `netmap` and `Libpcap`, where the rest of streaming analysis is performed.

Map & Filter offload. As shown in Table 1, WindFlow uses Map and Filter operators for one-to-one transformations and filtering of tuples, respectively. For transformations, offloading is viable if the Map operator performs *stateless* operations (i.e., transformations that do not require storing data into an internal data structure to be retrieved later by subsequent operators), that can be implemented using the eBPF instruction set. However, if transformations involve complex floating point operations or require SIMD instructions, they are deferred to the userspace pipeline.

For *stateful* operations, although it would be possible to store such information into eBPF maps and retrieve them later by reading such maps from userspace, we found it to be not as efficient as using userspace data structure due to limitations in Netronome's support for per-core eBPF maps. As a result, such operations are handled within userspace.

Regarding the Filter operators, offloading is feasible when supported by the eBPF instruction set and available eBPF map types. The offloaded operator filters tuples based on predefined policies, returning the `XDP_DROP` action for discarded tuples, as opposed to the default `XDP_PASS`. This approach optimizes the offloading of operators while considering their specific characteristics and hardware constraints, resulting in efficient stream data processing.

5. Use Cases

In this Section, we explore three practical use cases that demonstrate the versatility and efficiency of our offloading approach within the WindFlow framework. These use cases showcase how offloading specific stream processing tasks to Netronome SmartNICs can greatly enhance the performance and effectiveness of real-world applications.

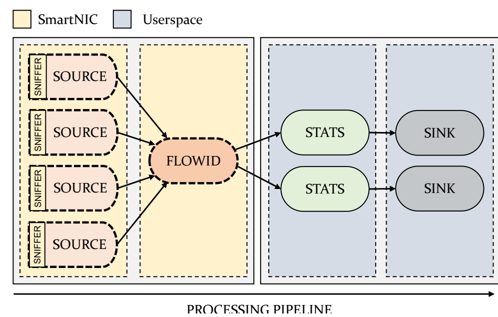

Figure 5: The flow meter application. In addition to the Source and Sink operators, we employ two Map operators (FlowID and Stats) to collect per-flow statistics.

5.1. Flow Meter

The *Flow Meter* application, inspired by [13], leverages the WindFlow architecture and mechanisms previously discussed to continuously update and maintain per-flow statistics, offering essential traffic analytics. These statistics include per-flow packet and byte counters, the number of observed flows, per-protocol counters for packet and flow volumes, and more.

Figure 5 provides an overview of the application's processing flow. Each parallel processing pipeline starts with one of the Source replicas, utilizing `nethuns`

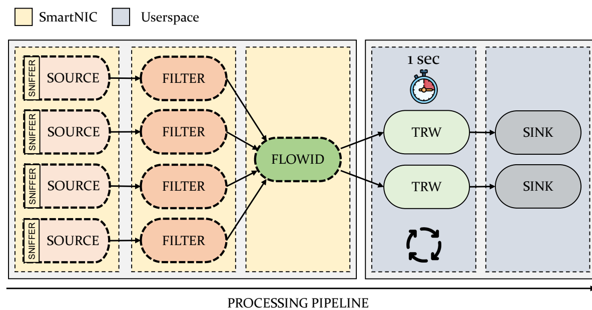

Figure 6: The Port Scan Detection application. It includes the Source and Sink operators, a Filter operator for incoming TCP tuples, a Map operator (FlowID) for tuple identification, and a Keyed Window operator for implementing the Threshold Random Walk (TRW) algorithm.

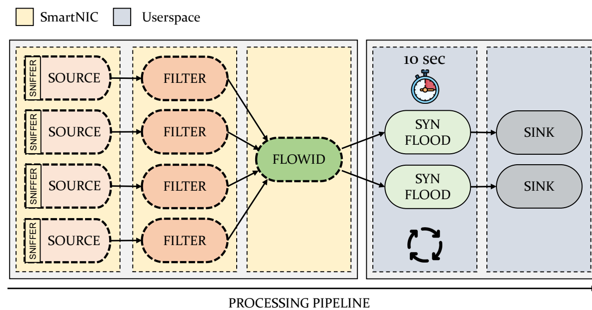

Figure 7: The SYN Flood Detection application. It includes the Source and Sink operators, a Filter operator for incoming TCP tuples, a Map operator (FlowID) for tuple identification, and a Keyed Window operator for the SYN Flood detection algorithm.

sockets [4] for accelerated packet capture. The Source operators perform three crucial tasks:

- *Parsing* the packet headers of interest
- *Extracting* tuples from the accessible channels
- *Forwarding* them to the next stage, which is the FlowID operator

In the FlowID stage, a unique FlowID is computed for each incoming tuple based on the canonical 5-tuple <src IP, dst IP, src port, dst port, protocol>. This FlowID serves as the key for distributing tuples among the replicas of the subsequent stage, the Stats operator.

The Stats operator, implemented as a WindFlow Map, continuously updates and maintains per-flow statistics. Finally, the Sink operator serves as the endpoint for the tuple stream, allowing the accumulated statistics to be exported or visualized for analysis.

In our offloaded architecture, the Source and FlowID operators are implemented as eBPF/XDP programs running on the SmartNIC's hardware eBPF engine, while the Stats operator remains in userspace to maximize performance.

5.2. Port Scan Detection

The *Port Scan Detection* application focuses on identifying suspicious network activity, specifically port scanning attempts by malicious attackers. It employs the Threshold Random Walk (TRW) algorithm, an online detection algorithm that identifies malicious remote hosts by modeling accesses to local IP addresses as a random walk on one of two stochastic processes, corresponding respectively to the access patterns of benign remote hosts and malicious ones [25].

Figure 6 illustrates the processing flow of this application. Similar to the other applications, parallel processing pipelines begin with the Source replicas, which parse headers of interest, including src IP, dst IP, and TCP header information such as TCP flags and src/dst port. In the subsequent Filter stage, packets with TCP `SYN` or `SYN-ACK` flags are filtered since they are the focus of the application. The FlowID stage computes a unique FlowID for each incoming tuple. For packets with the `SYN` flag, FlowID is computed using the `jhash` algorithm on <src IP, dst IP, dst port>, while for packets with `SYN-ACK` flags, it is computed on <dst IP, src IP, src port>.

The Source, Filter, and FlowID operators are entirely offloaded to the SmartNIC, and the final tuples are forwarded using `nethuns` sockets to the remaining part of the WindFlow pipeline in userspace.

Within the WindFlow pipeline, the TRW algorithm runs inside a Keyed Window operator, scanning all received tuples every second and generating alerts when the number of connections exceeds a configured threshold. Importantly, while the TRW component is not offloaded, it benefits from offloading as all necessary information has been extracted in the hardware pipeline by the previous operators.

5.3. SYN Flood Detection

The *SYN Flood Detection* application serves the purpose of identifying `SYN` flood attacks by monitoring and counting the number of incomplete TCP handshakes within a defined time interval [52]. In this context, an incomplete TCP handshake is characterized by the presence of a `SYN` packet followed by a `SYN-ACK` packet, each with corresponding sequence and acknowledge numbers, but lacking the subsequent `ACK` packet to complete the handshake.

Similar to previous applications, the processing pipeline for SYN Flood Detection, as depicted in Figure 7, begins with the Source operator, responsible for packet parsing and tuple creation. These tuples then

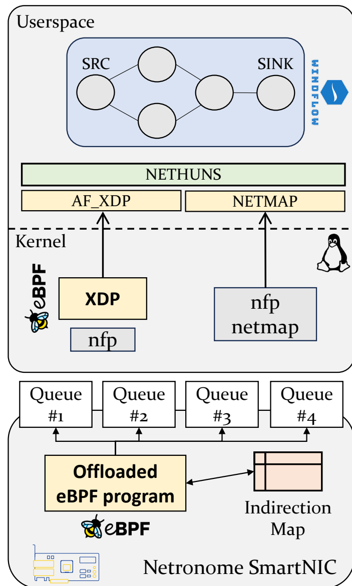

Figure 8: Test environment setup.

pass through the FILTER operator for further processing. In the FLOWID stage, a unique FLOWID is computed for each incoming tuple. To ensure consistency in flow identification for bidirectional traffic, we employ the algorithm described in [33]. The flow's key is defined as:

$$key = \{min(IpSrc, IpDest), max(IpSrc, IpDest), Proto,$$
$$min(PortSrc.PortDest), max(PortSrc, PortDest)\} \quad (1)$$

This key is then hashed using the `jhash` algorithm to generate the unique FLOWID. Additionally, two flags, *ip reverse* (`ipRev`) and *port reverse* (`portRev`), are added to the generated tuple to indicate the direction of the flow correctly.

On the userspace side, the SYN Flood detection algorithm is implemented within a KEYED WINDOW operator. This algorithm counts incomplete handshakes every 10 seconds and compares the count to a user-defined threshold to generate alerts.

6. Experimental evaluation

This section presents a series of experiments designed to assess the impact of our offloading technique. Our testbed comprises two identical machines featuring Xeon E5-1660 processors with 8 cores, operating at 3.0 GHz, equipped with 20 MiB of L3 cache and 32 GiB of DDR4 2133 MT/s DRAM. One machine is equipped with a Netronome Agilio CX 2x40Gbps SmartNIC, while the other features a 40 Gbps Intel XL710 NIC. We've disabled frequency scaling and hyperthreading since all WindFlow experiments utilize non-blocking queues, ensuring that all active cores run at maximum utilization. Both servers run Ubuntu 22.04 and kernel v5.19.0.

For the tests with the offload we configure an eBPF offloaded map (called `indirection table`) to redirect flows to a set of Receive Side Scaling (RSS) queues depending on the experiment, forcing the applications to be executed on a specific set of CPU cores.

To generate traffic and report throughput results, we employ `pktgen` [10] with DPDK v20.11.0. Additionally, we use the DPDK burst replay tool [44] to replay various packet traces. Unless stated otherwise, we report the average throughput across five runs of each benchmark, measured at the minimum packet size of 64 bytes. Each benchmark involves measuring the throughput of the streaming application using two distinct I/O frameworks: AF_XDP [7] and netmap [45]. Our approach relies on the `nethuns` [4] library, which abstracts the underlying I/O framework, ensuring that the userspace application remains unaltered.

When using AF_XDP, an additional eBPF/XDP program is loaded onto the host kernel. Its purpose is to direct frames to specific AF_XDP sockets [7]. Throughout our tests, we enable the `XDP_ZEROCOPY` flag, which has been recently introduced to the Netronome `nfp` driver [21]. This flag is important because it allows userspace applications using the AF_XDP sockets to receive and transmit packets without any copies from kernel to userspace. No modifications to the applications are needed, but the NIC driver needs to be modified to support zero-copy. In contrast, netmap does not necessitate such modifications. However, it does require changes to the driver to support the netmap memory model, which can subsequently be loaded as a kernel module. To perform out tests, we have accordingly modified the `nfp` driver to include netmap support[4].

6.1. Maximum throughput

To gauge the upper bounds of throughput, we start by measuring the maximum incoming throughput achievable using the Netronome SmartNIC, employing both the netmap and the AF_XDP frameworks.

In our test scenarios, we generate traffic from the Intel XL710 NIC, consisting of 32 flows of minimally-sized UDP packets, achieving the highest attainable aggregate packet rate of approximately 42Mpps within our

[4]`https://github.com/luigirizzo/netmap/tree/nfp`. Only the RX part of the driver is currently implemented. This is sufficient for the purposes of the paper.

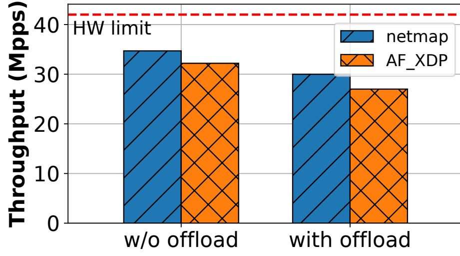

Figure 9: Baseline performance comparison of netmap and AF_XDP when using all available 8 hardware RX queues. When loading the eBPF program on the SmartNIC the throughput slightly decrease to ~30Mpps for *netmap* and 27Mpps for *AF_XDP*.

testbed. It's crucial to note that this rate serves as a hardware bottleneck, a characteristic acknowledged by Intel in their documentation [6, 11].

Figure 9 shows the results obtained from these baseline tests. When we employ the netmap `pkt-gen` receiver in busy-waiting mode, we observe a maximum incoming throughput of 34.7 Mpps, which remains the same regardless of the number of hardware RX queues employed. On the other hand, when running the AF_XDP `xdpsock` sample application [43], we achieve a maximum incoming throughput of 24.7 Mpps with a single hardware RX queue, which increases to 32.2 Mpps when all available hardware RX queues are used. These throughput numbers align with those reported in previous research [42] and the `nethuns` documentation [4].

The introduction of the eBPF tuple-parsing program into the NIC introduces some overhead. The maximum throughput for netmap drops to approximately 30 Mpps, while for AF_XDP, it drops to 27 Mpps. We identified that a significant portion of this overhead is attributed to the hash function calculations performed in the offloaded program. This hash function calculation is used both for the FlowID offload (as explained in §4.1) and for directing the flow to the appropriate RX queue using the `indirection` map.

Despite the reduction in achievable maximum throughput due to the offloaded program, the subsequent sections will highlight the substantial benefits of offloading for specific use cases.

6.2. *Raw tuple-generation test*

Here we describe a series of tests aimed at understanding the effects of the offloading for the various use cases presented above. We start from the simplest conceivable WindFlow application: an *empty pipeline* application consisting of just the two mandatory operators (a Source and a Sink), each one consuming a core per replica. In the non-offloaded case, the Source node receives the incoming packets and creates the corresponding tuples; in the offloaded case, the Source receives the tuples directly from the NIC and forwards them by pointer, without ever touching them. In both cases the Sink recycles the tuples' memory back to the Source.

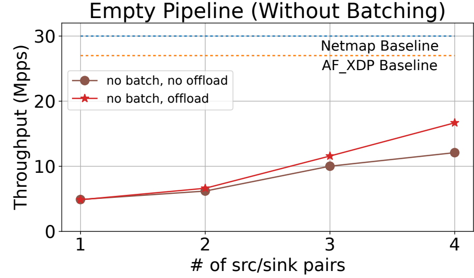

Figure 10: Throughput of the *empty pipeline* application with no batching. Results are the same for both *netmap* and *AF_XDP*.

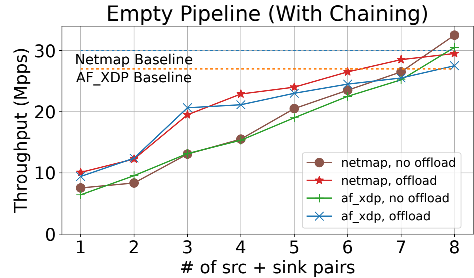

Figure 11: Throughput of the *empty pipeline* application with chaining. Results are the same for both *netmap* and *AF_XDP*.

Figure 10 shows the throughput of this application with and without offloading. We can see that, even when using all the 8 available cores (4 for the Source and 4 for the Sink), the application can process only less than half of the maximum achievable throughput. Offloading the source operator in the Netronome can slightly improve the packet rate, signaling that tuple creation may contribute to the bottle-neck. However, a much larger source of overhead comes from the communication overhead between the two operators running in two separate cores. This bottleneck can be removed in a couple of ways in the WindFlow framework: *(i)* by forwarding tuples in batches to amortize the communication cost, or *(ii)* by chaining the operators so that they are run one after the other in the same thread (and therefore the same core).

Figure 11 shows the effect of enabling operator chaining. We can see that chaining is also effective at removing the communication overhead. However, it is not available for all possible pairs of operators and, if the chained operators are close to saturation, it may even have a negative impact. Note that the non-offloaded case performs better than the offloaded one for 8 sources.

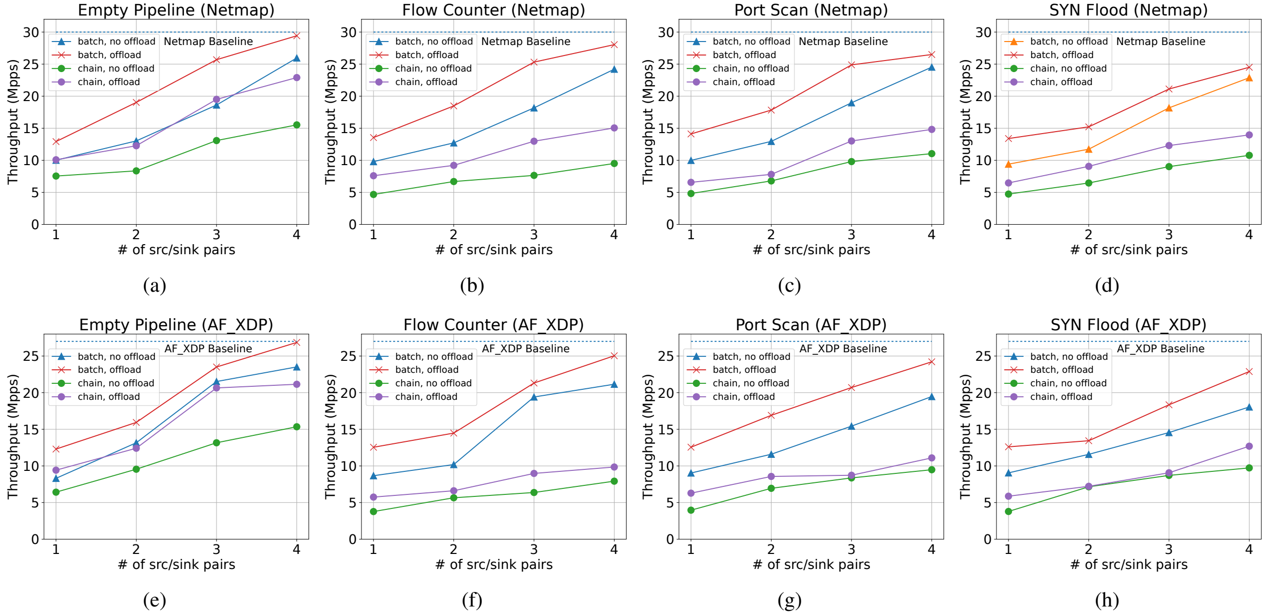

Figure 12: Throughput in millions of packets per second (Mpps) for four distinct WindFlow applications. For each application, we present the throughput results when using both the *netmap* and *AF_XDP* packet I/O frameworks. Additionally, we examine the impact of enabling WindFlow's batching and chaining features, both with and without our proposed offloading mechanism.

This is a direct effect of the different hardware limits described in Section 6.1 above.

Finally, figures 12a and 12e shows how the throughput, for both netmap and AF_XDP, changes when batching is enabled. Now the effect of source-offloading is much more visible and we get very close to the baseline limit with 3 sources.

6.3. *Use cases*

In this final series of experiments, we evaluate the throughput of the applications introduced in Section 5, considering both complete and partial operator offloading strategies. We present results for both the netmap and AF_XDP frameworks, with either batching or chaining enabled. Based on the findings in Section 6.2, we focus on reporting results with batching enabled.

Flow Meter. The Flow Meter application comprises a pipeline of four operators, except in the offloaded scenario where only three operators are used. This is because the FlowID operator runs entirely within the Netronome. Each operator can be assigned an independent degree of parallelism. We denote the degrees of parallelism for the four operators as n-m-p-q, in order. In the offloaded case we use n-m-p. Since each operator instance uses a full core, we must have $n+m+p+q \leq 8$ for the non offloaded cases, and $n+m+p \leq 8$ in the offloaded one, to avoid inefficient time-sliced execution.

Figures 12b and 12f illustrate the results. Offloading tuple creation significantly alleviates the Source operator's workload. By offloading tuple generation, the Source no longer needs to process every incoming tuple, mitigating the potential for cache misses. By offloading both the Source and FlowID operators, we achieve throughput close to the baseline for both netmap and AF_XDP when batching is enabled. This corresponds to a nearly 1.5× increase in throughput compared to the non-offloaded case.

Port Scan & Syn Flood. These applications introduce the use of another operator, the Keyed Window operator, as part of the WindFlow streaming processing pipeline. However, complete offloading of this operator in the Netronome card is not possible due to current hardware limitations. In both applications, to reduce the processing load on the userspace application, we completely offload the Source, FlowID and Filter operators. Partial offloading of the window-based operator involves parsing and calculating the `SYN` and `SYN-ACK` for a given flow and passing this information in the newly created tuple. This tuple is then received by the userspace operator without the need for redundant calculations.

Figures 12c,12g,12d, and 12h exhibit similar trends. The *Port Scan* application shows slightly lower performance than the *Flow Meter* due to the greater complexity of the hardware offloaded program, which involves more intricate operations. Nonetheless, the performance improvement from offloading is evident, re-

sulting in approximately a 25% improvement when using three SOURCE operators.

Similarly, the *Syn Flood* application demonstrates a comparable pattern, with a further decrease in performance due to the execution of a more complex algorithm associated with key ordering used in the SmartNIC to generate a single flow key for flows from both directions. However, the benefits of hardware offloading are clear, yielding improvements ranging from 10% to approximately 45% (netmap, one SOURCE/SINK).

7. Related Work

The cooperation between hardware and software for traffic analysis is certainly not new and many solutions have been proposed since the time of network processors [15, 49]. In particular, some of the ideas proposed here could be found in [15], where in a different scenario the Intel IXP2400 was used as the first stage of a monitoring system to capture and strip the relevant headers from network packets. In this paper, we focus on SmartNICs, which in turn have already been proposed for offloading traffic processing (e.g., in [14, 35]). However, to the best of our knowledge, very little has been done for offloading stream processing operations. More generally, there is still relatively little work focusing on performance acceleration on heterogeneous systems.

A notable (and nearly unique) attempt to combine the expressiveness of DaSP system with hardware class performance for stream processing in the network domain is Sonata [18, 19]. Sonata stands out as a hybrid framework that combines a P4 programmable switch with stream processors. The workflow begins with the submission of monitoring queries through a declarative interface, which are then distributed between a stream processor (implemented with Spark Streaming [3]) and a programmable switch. By offloading a substantial number of queries directly to the switch, Sonata effectively prevents overwhelming the stream processor and enables significantly high packet rates.

Instead, more has been done with GPUs. G-Storm [5] served as a GPU-enabled parallel system based on Storm. It harnessed the vast computing power of GPUs for high-throughput online stream data processing. Saber [28] implemented a hybrid high-performance relational stream processing engine for CPUs and GPGPUs. Designed to execute window-based streaming SQL queries in a data-parallel fashion, Saber utilized all available CPU and GPGPU cores. F-Storm [50] stood out as a general-purpose distributed stream processing system for Edge servers, leveraging on-board PCIe-based FPGAs for acceleration. The authors demonstrated that, compared to Storm, F-Storm significantly improved the speed of algebraic computations and grep applications while reducing latency.

FineStream [53, 54] represented a stream processing engine based on a CPU-GPU integrated architecture, targeting the efficient handling of multiple queries in both static and dynamic streams. Recently, Liu et al. introduced FineQuery [30], a fine-grained query processing engine capable of efficient query processing on CPU-GPU integrated edge devices. FineQuery maximizes the advantages of edge device architectural features and query characteristics through fine-grained workload scheduling between the CPU and the GPU.

8. Conclusions

In this paper we show that SmartNICs are an affordable solution to accelerate standard network monitoring analysis tasks. We started from WindFlow, a state-of-the-art framework for stream processing, and demonstrated that it can be repurposed to analyze network data. We then implemented a custom logic to be run on a standard commodity SmartNIC, the Netronome Agilio CX, that partially offload the compute required by WindFlow.

To ensure the generality of our approach we developed the NIC logic in eBPF so that it can be ported to any NIC or hardware accelerator supporting this programming paradigm. Finally, we implemented a number of traffic analysis use-cases: (1) flow metering; (2) port scan detection; and (3) SYN flood attack detection. We show that our approach can ingest and analyze up to 40% more traffic compared to a pure software approach.

References

[1] Apache, 2014. Storm. [Online]. Available: `https://storm.apache.org/`. Accessed: October 25, 2023.

[2] Apache, 2015a. Flink. [Online]. Available: `https://flink.apache.org/`. Accessed: October 25, 2023.

[3] Apache, 2015b. Spark Streaming. [Online]. Available: `https://spark.apache.org/streaming/`. Accessed: October 25, 2023.

[4] Bonelli, N., Vigna, F.D., Fais, A., Lettieri, G., Procissi, G., 2022. Programming socket-independent network functions with nethuns. SIGCOMM Comput. Commun. Rev. 52, 35–48. URL: `https://doi.org/10.1145/3544912.3544917`, doi:`10.1145/3544912.3544917`.

[5] Chen, Z., Xu, J., Tang, J., Kwiat, K.A., Kamhoua, C.A., Wang, C., 2018. Gpu-accelerated high-throughput online stream data processing. IEEE Transactions on Big Data 4, 191–202. doi:`10.1109/TBDATA.2016.2616116`.

[6] Corporation, I., 2014. Intel xl710 controller brief. [Online]. Available: https://lafibre.info/images/materiel/201408_intel_xl710_controller_brief.pdf. Accessed: October 25, 2023.

[7] Documentation, T.L.K., 2023a. AF_XDP. [Online]. Available: https://www.kernel.org/doc/html/next/networking/af_xdp.html. Accessed: October 25, 2023.

[8] Documentation, T.L.K., 2023b. BPF Maps. [Online]. Available: https://www.kernel.org/doc/html/v5.18/bpf/maps.html. Accessed: October 25, 2023.

[9] Documentation, T.L.K., 2023c. XDP RX Metadata. [Online]. Available: https://docs.kernel.org/networking/xdp-rx-metadata.html. Accessed: October 25, 2023.

[10] DPDK, 2018. Pktgen Traffic Generator Using DPDK. [Online]. Available: http://dpdk.org/git/apps/pktgen-dpdk. Accessed: October 25, 2023.

[11] Emmerich, P., Gallenmüller, S., Raumer, D., Wohlfart, F., Carle, G., 2015. Moongen: A scriptable high-speed packet generator, in: Proceedings of the 2015 Internet Measurement Conference, Association for Computing Machinery, New York, NY, USA. p. 275–287. URL: https://doi.org/10.1145/2815675.2815692, doi:10.1145/2815675.2815692.

[12] Fais, A., Antichi, G., Giordano, S., Lettieri, G., Procissi, G., 2022. Mind the cost of telemetry data analysis, in: Proceedings of the SIGCOMM '22 Poster and Demo Sessions, Association for Computing Machinery, New York, NY, USA. p. 22–24. URL: https://doi.org/10.1145/3546037.3546052, doi:10.1145/3546037.3546052.

[13] Fais, A., Lettieri, G., Procissi, G., Giordano, S., 2021. Towards scalable and expressive stream packet processing, in: IEEE Global Communications Conference, GLOBECOM 2021, Madrid, Spain, December 7-11, 2021, IEEE. pp. 1–6. URL: https://doi.org/10.1109/GLOBECOM46510.2021.9685436, doi:10.1109/GLOBECOM46510.2021.9685436.

[14] Feng, Y., Panda, S., Kulkarni, S.G., Ramakrishnan, K.K., Duffield, N., 2020. A smartnic-accelerated monitoring platform for in-band network telemetry, in: 2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN, pp. 1–6. doi:10.1109/LANMAN49260.2020.9153279.

[15] Ficara, D., Giordano, S., Oppedisano, F., Procissi, G., Vitucci, F., 2008. A cooperative pc/network-processor architecture for multi gigabit traffic analysis, in: 2008 4th International Telecommunication Networking Workshop on QoS in Multiservice IP Networks, pp. 123–128. doi:10.1109/ITNEWS.2008.4488141.

[16] Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A., Chung, E., Chandrappa, H.K., Chaturmohta, S., Humphrey, M., Lavier, J., Lam, N., Liu, F., Ovtcharov, K., Padhye, J., Popuri, G., Raindel, S., Sapre, T., Shaw, M., Silva, G., Sivakumar, M., Srivastava, N., Verma, A., Zuhair, Q., Bansal, D., Burger, D., Vaid, K., Maltz, D.A., Greenberg, A., 2018. Azure accelerated networking: SmartNICs in the public cloud, in: 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), USENIX Association, Renton, WA. pp. 51–66. URL: https://www.usenix.org/conference/nsdi18/presentation/firestone.

[17] Guo, C., Yuan, L., Xiang, D., Dang, Y., Huang, R., Maltz, D., Liu, Z., Wang, V., Pang, B., Chen, H., Lin, Z.W., Kurien, V., 2015. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis, in: Special Interest Group on Data Communication (SIGCOMM), ACM. doi:10.1145/2785956.2787496.

[18] Gupta, A., et al., 2016. Network Monitoring as a Streaming Analytics Problem, in: 15th ACM Workshop on HotNets '16, ACM. p. 106–112. URL: https://doi.org/10.1145/3005745.3005748, doi:10.1145/3005745.3005748.

[19] Gupta, A., et al., 2018. Sonata: Query-Driven Streaming Network Telemetry, in: Proceedings of the 2018 Conference of the ACM SIGCOMM '18, ACM. p. 357–371. URL: https://doi.org/10.1145/3230543.3230555, doi:10.1145/3230543.3230555.

[20] Høiland-Jørgensen, T., Brouer, J.D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., Miller, D., 2018. The express data path: Fast programmable packet processing in the operating system kernel, in: Proc. of CoNEXT '18, Association for Computing Machinery, New York, NY, USA. p. 54–66. URL: https://doi.org/10.1145/3281411.3281443, doi:10.1145/3281411.3281443.

[21] Horman, S., 2022. Add AF_XDP zero-copy support for NFP. [Online]. Available: https://lore.kernel.org/netdev/20220304102214.25903-1-simon.horman@corigine.com/. Accessed: October 25, 2023.

[22] Intel, 2020. In-band Network Telemetry Detects Network Performance Issues. [White Paper]. Available: https://builders.intel.com/docs/networkbuilders/in-band-network-telemetry-detects-network-performance-issues.pdf. Accessed: October 25, 2023.

[23] IO Visor Project, 2023. eBPF - Technology. [Online]. Available: https://www.iovisor.org/technology/ebpf. Accessed: October 25, 2023.

[24] Jung, J., Paxson, V., Berger, A., Balakrishnan, H., 2004a. Fast portscan detection using sequential hypothesis testing, in: IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004, pp. 211–225. doi:10.1109/SECPRI.2004.1301325.

[25] Jung, J., Paxson, V., Berger, A., Balakrishnan, H., 2004b. Fast portscan detection using sequential hypothesis testing, in: IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004, pp. 211–225. doi:10.1109/SECPRI.2004.1301325.

[26] Khandelwal, A., Agarwal, R., Stoica, I., 2019. Confluo: Distributed Monitoring and Diagnosis Stack for High-Speed Networks, in: Networked Systems Design and Implementation (NSDI), USENIX.

[27] Kicinski, J., Viljoen, N., 2018. Xdp hardware offload: Current work, debugging and edge cases. URl: https://legacy. netdevconf. info/2.2/papers/viljoen-xdpoffload-talk. pdf .

[28] Koliousis, A., Weidlich, M., Castro Fernandez, R., Wolf, A.L., Costa, P., Pietzuch, P., 2016. Saber: Window-based hybrid stream processing for heterogeneous architectures, in: Proceedings of the 2016 International Conference on Management of Data, Association for Computing Machinery, New York, NY, USA. p. 555–569. URL: https://doi.org/10.1145/2882903.2882906, doi:10.1145/2882903.2882906.

[29] Li, Y., Miao, R., Kim, C., Yu, M., 2016. FlowRadar: A Better NetFlow for Data Centers, in: Networked Systems Design and Implementation (NSDI), USENIX.

[30] Liu, J., Zhang, F., Li, H., Wang, D., Wan, W., Fang, X., Zhai, J., Du, X., 2022. Exploring query processing on cpu-gpu integrated edge device. IEEE Transactions on Parallel and Distributed Systems 33, 4057–4070. doi:10.1109/TPDS.2022.3177811.

[31] Mencagli, G., Torquati, M., Cardaci, A., Fais, A., Rinaldi, L., Danelutto, M., 2021. WindFlow: High-Speed Continuous Stream Processing with Parallel Building Blocks, in: Transactions on Parallel and Distributed Systems (TPDS), Volume: 32, Issue: 11, IEEE. doi:10.1109/TPDS.2021.3073970.

[32] Miano, S., Bertone, M., Risso, F., Bernal, M.V., Lu, Y., Pi, J., Shaikh, A., 2019a. A service-agnostic software framework for fast and efficient in-kernel network services, in: Symposium on Architectures for Networking and Communications Systems

(ANCS), IEEE. doi:10.1109/ancs.2019.8901880.

[33] Miano, S., Bertrone, M., Risso, F., Bernal, M.V., Lu, Y., Pi, J., 2019b. Securing linux with a faster and scalable iptables. SIGCOMM Comput. Commun. Rev. 49, 2–17. URL: https://doi.org/10.1145/3371927.3371929, doi:10.1145/3371927.3371929.

[34] Miano, S., Chen, X., Basat, R.B., Antichi, G., 2023. Fast in-kernel traffic sketching in ebpf. SIGCOMM Comput. Commun. Rev. 53, 3–13. URL: https://doi.org/10.1145/3594255.3594256, doi:10.1145/3594255.3594256.

[35] Miano, S., Doriguzzi-Corin, R., Risso, F., Siracusa, D., Sommese, R., 2019c. Introducing smartnics in server-based data plane processing: The ddos mitigation use case. IEEE Access 7, 107161–107170. doi:10.1109/access.2019.2933491.

[36] Miano, S., Sanaee, A., Risso, F., Rétvári, G., Antichi, G., 2022. Domain specific run time optimization for software data planes, in: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Association for Computing Machinery, New York, NY, USA. p. 1148–1164. URL: https://doi.org/10.1145/3503222.3507769, doi:10.1145/3503222.3507769.

[37] Narayana, S., Sivaraman, A., Nathan, V., Goyal, P., Arun, V., Alizadeh, M., Jeyakumar, V., Kim, C., 2017. Language-Directed Hardware Design for Network Performance Monitoring, in: Special Interest Group on Data Communication (SIGCOMM), ACM. doi:10.1145/3098822.3098829.

[38] Netronome, 2018. eBPF Offload Getting Started Guide. [Online]. Available: https://www.netronome.com/media/documents/UG_Getting_Started_with_eBPF_Offload.pdf. Accessed: October 25, 2023.

[39] Netronome, 2020. Agilio CX 2x40GbE. [Online]. Available: https://www.netronome.com/media/documents/PB_Agilio_CX_2x40GbE-7-20.pdf. Accessed: October 25, 2023.

[40] Netronome, 2023. Programmable RSS. GitHub repository. URL: [Online].Available:https://github.com/Netronome/bpf-samples/tree/master/programmable_rss.

[41] Parola, F., Miano, S., Risso, F., 2021. Providing telco-oriented network services with ebpf: the case for a 5g mobile gateway, in: International Conference on Network Softwarization (NetSoft), IEEE. doi:10.1109/netsoft51509.2021.9492571.

[42] Parola, F., Procopio, R., Querio, R., Risso, F., 2023. Comparing user space and in-kernel packet processing for edge data centers. SIGCOMM Comput. Commun. Rev. 53, 14–29. URL: https://doi.org/10.1145/3594255.3594257, doi:10.1145/3594255.3594257.

[43] Project, X., 2023. AF_XDP-example in bpf-examples. [Online]. Available: https://github.com/xdp-project/bpf-examples/tree/master/AF_XDP-example. Accessed: October 25, 2023.

[44] Ribas, J., 2019. DPDK burst replay tool. [Online]. Available: https://github.com/FraudBuster/dpdk-burst-replay. Accessed: October 25, 2023.

[45] Rizzo, L., 2012. Netmap: a novel framework for fast packet i/o, in: Proc. of USENIX ATC 2012, USENIX Association. pp. 1–12. doi:10.1109/infcom.2012.6195638.

[46] Silicon Valley Business Journal, 2023. Intel halts development of Tofino switch chips. [Online]. Available: https://www.bizjournals.com/sanjose/news/2023/01/26/intel-halts-development-of-tofino-switch-chips.html. Accessed: October 25, 2023.

[47] Starovoitov, A., 2021. bpf: Introduce bpf timers. [Online]. Available: https://lore.kernel.org/bpf/20210715005417.78572-4-alexei.starovoitov@gmail.com/. Accessed: October 25, 2023.

[48] Van Tu, N., Hyun, J., Kim, G.Y., Yoo, J.H., Hong, J.W.K., 2018. INTCollector: A High-performance Collector for In-band Network Telemetry, in: Conference on Network and Service Management (CNSM), IEEE.

[49] Wolf, T., Ramaswamy, R., Bunga, S., Yang, N., 2006. An architecture for distributed real-time passive network measurement, in: 14th IEEE International Symposium on Modeling, Analysis, and Simulation, pp. 335–344. doi:10.1109/MASCOTS.2006.11.

[50] Wu, S., Hu, D., Ibrahim, S., Jin, H., Xiao, J., Chen, F., Liu, H., 2019. When fpga-accelerator meets stream data processing in the edge, in: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pp. 1818–1829. doi:10.1109/ICDCS.2019.00180.

[51] Yuan, Y., Lin, D., Mishra, A., Marwaha, S., Alur, R., Loo, B.T., 2017a. Quantitative network monitoring with netqre, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, Association for Computing Machinery, New York, NY, USA. p. 99–112. URL: https://doi.org/10.1145/3098822.3098830, doi:10.1145/3098822.3098830.

[52] Yuan, Y., Lin, D., Mishra, A., Marwaha, S., Alur, R., Loo, B.T., 2017b. Quantitative network monitoring with netqre, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, Association for Computing Machinery, New York, NY, USA. p. 99–112. URL: https://doi.org/10.1145/3098822.3098830, doi:10.1145/3098822.3098830.

[53] Zhang, F., Yang, L., Zhang, S., He, B., Lu, W., Du, X., 2020. FineStream: Fine-Grained Window-Based stream processing on CPU-GPU integrated architectures, in: 2020 USENIX Annual Technical Conference (USENIX ATC 20), USENIX Association. pp. 633–647. URL: https://www.usenix.org/conference/atc20/presentation/zhang-feng.

[54] Zhang, F., Zhang, C., Yang, L., Zhang, S., He, B., Lu, W., Du, X., 2021. Fine-grained multi-query stream processing on integrated architectures. IEEE Transactions on Parallel and Distributed Systems 32, 2303–2320. doi:10.1109/TPDS.2021.3066407.

[55] Zhou, Y., Sun, C., Liu, H.H., Miao, R., Bai, S., Li, B., Zheng, Z., Zhu, L., Shen, Z., Xi, Y., Zhang, P., Cai, D., Zhang, M., Xu, M., 2020. Flow Event Telemetry on Programmable Data Plane, in: Special Interest Group on Data Communication (SIGCOMM), ACM. doi:10.1145/3387514.3406214.