If you cite this paper, please use the ACM CoNEXT reference: Nikita Tyunyayev, Clément Delzotti, Haggai Eran, and Tom Barbette. 2025. ASNI: Redefining the Interface Between SmartNICs and Applications. Proc. ACM Netw. 3, CoNEXT2, Article 9 (June 2025), 22 pages. https://doi.org/10.1145/3730966

ASNI: Redefining the Interface Between SmartNICs and Applications

NIKITA TYUNYAYEV, UCLouvain, Belgium
CLÉMENT DELZOTTI, UCLouvain, Belgium
HAGGAI ERAN, NVIDIA, USA
TOM BARBETTE, UCLouvain, Belgium

With a budget of 300 CPU cycles per packet, Network Functions Virtualisation (NFV) scenarios require efficient packet exchange mechanisms between NICs and CPUs. However, even the latest kernel bypass techniques, such as DPDK, can eat up 20% of the budget. This paper investigates the strain on CPUs due to escalating network speeds and the ability to exchange more packet metadata due to NICs becoming more feature-fledged and "Smart". We advocate for NICs to adapt to software needs rather than rely on drivers as translators. We analyze different applications and compare multiple packet buffers and data organization models. We then develop an API for the datapath that will compile into different buffer management models according to the application's needs. Introducing ASNI, we explore the potential to tailor packet descriptors for different applications, offloading the driver's work from the datapath so that the application receives the packets precisely as it needs them. We advocate for a vision where only a minimal driver on the host is required, retrieving CPU cycles to applications instead. We propose a prototype implementation on NVIDIA BlueField-3 SoC. Evaluated in multiple NFV scenarios, it manages to serve 2.2x more traffic under the same loss ratio constraints than state-of-the-art solutions.

CCS Concepts: • **Networks** → **Middle boxes / network appliances**; **Network adapters**; *Network servers*.

Additional Key Words and Phrases: SmartNIC, high-speed packet processing, DPDK, packet descriptor

1 Introduction

Ever-increasing network speeds put pressure on CPUs, especially since the end of Dennard scaling and the complexity of scheduling packets to many-core CPU [2, 23]. For instance, at 100G, a 2.5GHz 16-core CPU has a budget of around 300 cycles per minimal-size packet, which stresses Network Function Virtualization (NFV) scenarios and network-intensive applications like key-value stores. To fulfill such requirements, an efficient way to exchange packets between the CPU and the Network Interface Controller (NIC) is needed.

At the heart of the exchange mechanisms between a NIC and the software environment lies the packet descriptor. Descriptors carry meta-information about the packet, such as its size, from the NIC to the driver through PCI Express. They precede the corresponding packets sent to the application using a different structure. Additionally, NFV frameworks such as [3, 12] require another transformation on top of this to get their own metadata format.

For a decade, kernel-bypass solutions like DPDK [14] and later XDP [55] have directly received packets in user space, avoiding the feature-rich but slow Linux stack [3, 50]. These user-space

Accepted preprint version for CoNEXT'25

Authors' Contact Information: Nikita Tyunyayev, UCLouvain, Louvain-la-Neuve, Belgium, nikita.tyunyayev@uclouvain.be; Clément Delzotti, UCLouvain, Louvain-la-Neuve, Belgium, clement.delzotti@uclouvain.be; Haggai Eran, NVIDIA, Santa Clara, USA, haggaie@nvidia.com; Tom Barbette, UCLouvain, Louvain-la-Neuve, Belgium, tom.barbette@uclouvain.be.

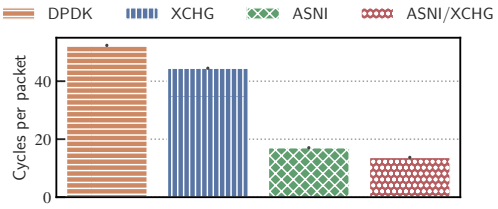


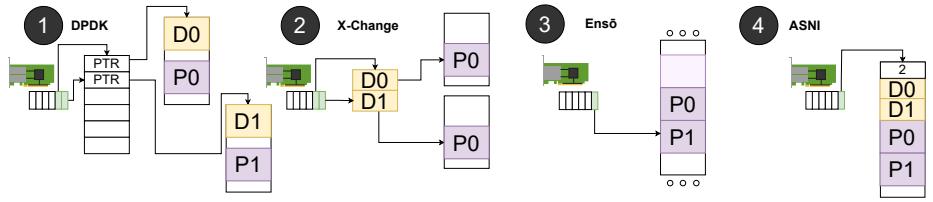
Fig. 1. Number of cycles taken to receive 64B packets using DPDK, X-Change, and ASNI over DPDK and X-Change

stacks allow customizing the network stack for the applications, but their abstractions still incur an overhead to support a wide range of NIC features. Modern NICs are hardly just about receiving and sending bytes. Commodity datacenter NICs are also in charge of packet classification, decapsulating outer and inner VLAN, tunneling offload with VXLAN/IPSec, timestamping, load-balancing between cores [51], tagging packets [25], computing checksums, and many more functions [24]. Therefore, the descriptors have grown over time to exchange metadata about features such as packet flow ID, tunnel ID, or checksum validity. For instance, DPDK in its first public version 1.2 used a 64 bytes *rte_mbuf*, which has grown to 128 bytes today to maintain all the metadata for those features [15]. XDP similarly added support for fragmented packets, RSS hash, and timestamp fields in the metadata. Moreover, it is not possible for an application to receive packets through any other means than the classical packetized interface, where the CPU exposes fixed-size buffers that are filled with incoming packets by the NICs. Applications that focus on throughput *could* prefer an interface to wait for packets and receive batches of packets in large contiguous buffers at the cost of a possible increase in latency.

Typical applications only require some of the fields and some of the features. The unnecessarily large structure must be filled, which will cause cache misses and reduce performance. The driver must comply with APIs, fill timestamps, verify if buffers are not shared before recycling them, and many more features the application *might* not need.

So inevitably, history repeats itself. TinyNF [47] proposes a minimal driver for the Intel 82599 10G NIC, which also prevents buffering and removes all features of modern NICs which are essential to many applications. Similarly, Ensō [52] proposes using a continuous buffer of payloads. This prevents buffering packets (including reordering), or forwarding packets without copying the packet payload. Ensō ignores the growing need for metadata exchange due to the increasing roles the NICs are taking today. In the same attempt to avoid the cost of an intermediate layer as with DPDK, recent high-speed oriented I/O frameworks like Caladan [16] use DPDK but also ship a more effective and hardware-specific descriptor. The Vector Packet Processing (VPP) [12] also took the same path [13] of directly implementing a shim driver for NVIDIA Mellanox NICs, avoiding DPDK transformations. PacketMill [11] proposes the *X-Change* model to merge the application and the driver using Link-time-optimization (LTO), avoiding transformation from the DPDK *rte_mbuf* to the application format. Figure 1 shows a motivating experiment where we count the number of cycles to receive packets, simply counting the total number of bytes. DPDK uses ~ 52 cycles per packet. X-Change avoids DPDK internals and directly accesses the NIC rings, using ~ 44 cycles per packet, which is still 14% of the 300 cycles budget.

Instead of relying on generic packet descriptors, we propose a fundamental change in how drivers are seen. We propose that the NIC should change according to the software, rather than



Zero-copy (payload)	✓	✓	✗	✓
Copies (descriptor)	2	1	0	0
Transformations	2	1	1	0
Application-space	✓	✓	✗	✓
Hardware offloads	✓	✓	✗	✓
DMA transfers / packet	O(1)	O(1)	O(1/N)	O(1/N)

Fig. 2. Various packet buffer models and their characteristics. N is the number of packets in a burst.

using the driver as a translator. The NIC should tailor descriptors for each specific application. Essentially, we push most of the driver data path to the NIC itself.

We therefore propose ASNI as an Application Specific Network Interface. ASNI uses DPDK for NIC initialization, but proposes its own interface to receive and transmit packets in batches. We compile the application according to its needs (batching, buffering, descriptor fields, prefetching header fields, etc.), to use a custom descriptor format and the corresponding SmartNIC pipeline. In Figure 1, ASNI hands out a data structure to the application that only needs to be cast as a pointer to an array of packet descriptors, leading to a 3.7x reduction in cycles per packet. Section 2 further describes the design of ASNI.

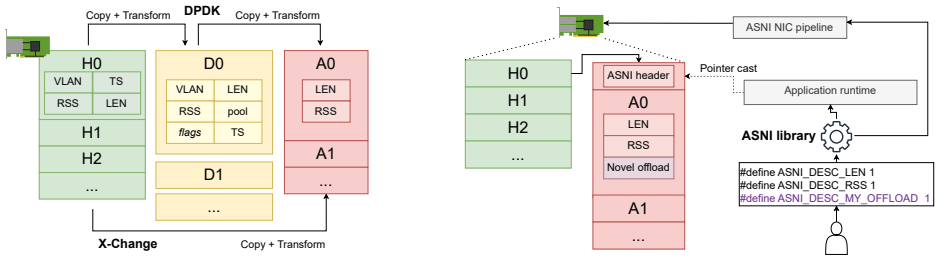
In this paper, we answer two key questions this novel concept raises. First, *what are the benefits for the host?* We show that avoiding most of the driver work of creating packet descriptors in the datapath improves the throughput by a 2.2x factor for a load balancer at the cost of a higher latency for lower loads in Section 5.7. Section 3 describes the ASNI library. Secondly, *can we prototype such a NIC with commodity hardware?* We propose a pipeline for a NIC capable of such preparation based on an NVIDIA BlueField-3 SmartNIC described in Section 4, and we explore the use of its built-in accelerators to improve the implementation. Section 5 evaluates the prototype with micro-benchmarks and some realistic applications. Section 6 mentions related works and Section 7 discusses how ASNI could be generalized to other platforms.

2 Processing models

In this section, we discuss the techniques prior work uses to receive data (payload) and metadata (descriptor) from the NIC. For clarity, we only describe the receive side here, where the NIC produces the descriptors and the CPU consumes them. Throughout our review, we note *ideas that we leverage* and *observation for improvements* that should be implemented in our design. We then explain our proposed approach for ASNI.

2.1 Traditional models

In traditional I/O frameworks, all packet buffers are received through independent transfers and are therefore scattered in memory (Fig. 2.❶ and ❷). Ensō [52] receives packets as a continuous stream (Fig. 2.❸). *Batched payloads transfer* are highly efficient, as the application directly receives the raw payload without any indirection layer, but only a producer/consumer notification mechanism. However, this indefinite stream of payload prevents the use of zero-copy since the payload must be copied to a TX contiguous buffer or use a shared RX/TX pipe in the case of Ensō when packets are processed in order. We need to *enable zero-copy forwarding* in all scenarios.



(a) DPDK and X-Change (b) ASNI

Fig. 3. Metadata copies and transformations with DPDK, X-Change, and ASNI from the NIC hardware (H), the descriptor (D), and the application (A).

Ensō makes the case that most information provided in the descriptor, such as the packet length, can be recomputed from the packet payload itself. As stated earlier, *modern NICs need to exchange much more packet metadata* that is sometimes not even part of the packet itself (timestamp, tunnel ID decapsulated in hardware, ...). We argue not only that packet descriptors are a necessity, but also that we should *pack many descriptors together*, especially when combined with minimized descriptors, so that data from multiple packets can be put into the same cache line.

2.2 Metadata transformation

Traditional network stacks like the Linux Kernel and DPDK run a NIC-specific driver to convert the hardware descriptor format to a standard format, offering an abstract indirection layer as shown in Figure 3a.

In DPDK, applications give an array of pointers to descriptors to be filled by the DPDK driver, and then packets are usually processed in batches. This leads to two copies and transformations of metadata, sometimes three, since the application might need a different descriptor format (e.g., VPP [12] or FastClick [3]). This is why X-Change [11] proposed eliminating the intermediate DPDK internal format, thereby *removing the need for any intermediate driver transformation.* Instead, the application links with functions that are called for every possible field that a driver might receive from the NIC (e.g. *set_vlan()*, *set_timestamp()*, ...). The application provides functions that are tailored to its descriptor format. Using link-time-optimization (LTO), functions *provided by the application* are inlined *in the driver*. Still, we would like to *avoid the hardware-application transformation.*

Modern NICs can precompute a good deal of information about incoming packets that the application can use to ease packet processing. A router can retrieve the VLAN that was stripped in hardware and the port from which the packet arrived directly from the descriptor. Monitoring applications can use the hash of a packet to compute per-flow statistics [59]. In the case of a tunnel packet, it is possible to retrieve the type of the tunnel directly from the descriptor, too. However, no application needs *all* of those together. We need to *enable tailored metadata transfer and layout.*

2.3 Indirection layer

Applications that rely on the packetized interface have to deal with an indirection layer. They have to first access a descriptor, fetch a pointer to the payload, and then access the payload. Therefore, multiple memory accesses are required to get to the payload. However, having indirection is *sometimes* useful, as it makes reordering and complex processing graphs possible.

Table 1. Classification of application regarding buffering and forwarding needs

	Buffering	Non-buffering
Forwarding	Schedulers, rate limiters, IPS withholding packets	NFV applications: firewall, NAT, LB, IPS using RST mechanism
Non-forwarding	Network stacks with reordering, e.g., TCP stack	RPC, RX-only applications for monitoring such as IDS

We observe that applications need different models, and no one-size-fits-all case exists. We classify applications in two axes: the buffering model, and whether they forward packets or only receive or create unrelated packets in response, summarized in Table 1.

Some applications need to buffer packets, like TCP stacks, which might reorder packets and delay the payload processing for a long time. For those, the indirection layer is necessary.

Some applications *run-to-completion (RTC) without buffering packets.* Monitoring applications can therefore simply access the pointer in the NIC rings, only updating consumers indexes from time to time. RTC applications which forward packets still need to swap buffers, so the buffer can be placed in the TX ring.

2.4 ASNI

ASNI relies on two components: one running on the SmartNIC to prepare tailored data structures for each application, and a compile-time layer to abstract ASNI in the I/O datapath of the application running on the host. We explain here the overall processing model of ASNI, then the application part in Section 3, and the prototype NIC in Section 4.

First, the NIC builds a tailored descriptor for each packet, which is then batched with payloads and sent in a single transaction towards the host as shown in Figure 2.❹. The application running on the host receives a single large memory transfer containing a burst of packets, starting with an array of descriptors *ready to be cast*. The application can work directly without any further transformation. The array does not have an indirection layer, so the cache efficiently prefetches all descriptors. However, a reference counting mechanism *can* be used to enable buffering and forwarding at the batch level.

2.4.1 Tailoring the datapath. At compilation time, the application specifies which information to extract from the packet itself, and the packet metadata (RSS hash, the result of rule classification, ...) with compilation flags as shown in Figure 3b. In our prototype, we use a C header file provided by the application developer that enables possible metadata fields in the descriptor used by both the application and the SmartNIC pipeline. We discuss possible alternative implementations, such as using P4 to describe the descriptor in Section 7.

2.4.2 Batching. We observe many recent works advocating for packet batching [3, 17, 27, 29]. A key idea behind ASNI is that most Kernel Bypass Networking (KBN) applications call their receive-API with an argument to receive, in general, up to 32 packets. ASNI delivers all packets *that are siting in the queue anyway* as a whole. This principle has the added benefit of adapting the batching size dynamically. When the throughput increases, the queues fill up, and the NIC sends bigger batches to the host, making the host processing more efficient as shown in Sections 5.4 and 5.7. ASNI also supports modes where it waits a maximum of S cycles to receive P packets. S and P are configurable.

2.4.3 Precomputation. Applications need their own per-packet space to keep metadata. Similarly, different applications need different fields from the hardware. While some may only require a pointer to the buffer and the buffer length, others may need many more fields. Applications can request information extracted from the packet stored in the descriptor. In addition to fields already

```
struct rte_mbuf* pkts[32];
int n =
  rte_eth_rx_burst(port, queue,
                   pkts);
for (i = 0 ... n)
  do_something(pkts[i],
               pkts[i]->buffer);
```

(a) DPDK application

```
ASNI_init();
...
struct descriptor* desc;
ASNI_for_each_packet(port, queue,
                     desc, buffer) {
  do_something(desc, buffer);
}
ASNI_end_for_each;
```

(b) ASNI application using our wrapper

```
struct xchgs* xchgs[32];
int n =
  rte_eth_rx_burst_xchg(port, queue, xchgs);
for (i = 0 ... n) {
  ASNI_header* header = xchgs[i]->data;
  uint8_t ndesc = header->ndesc;
  struct descriptor* descs = header + 1;
  char* payload = descs[ndesc + 1];
  for (j= 0 ... ndesc) {
    struct descriptor* desc = descs[j];
```

(c) Main pseudocode of the ASNI_for_each_packet macro (version over X-Change [11])

Fig. 4. Pseudocode of a usual DPDK application, the ASNI ported version and the main code of the ASNI wrapper

present in the descriptor and the packet, ASNI enables the construction of new custom fields for the application. For instance, adding the source IP address in the descriptor to let the application start working while the payload is fetched from memory to cache takes 10 LoC. ASNI is therefore future-proof, enabling easy support for new metadata from novel hardware offloads.

It is also worth mentioning that computing fields that require access to the payload on the NIC is not free and consumes the PCIe bandwidth available to the application while reducing the number of cycles required to process a packet. On the other hand, batching packets and descriptors saves a great deal of PCIe transactions. This tradeoff is evaluated in Section 5.6.

3 ASNI application library

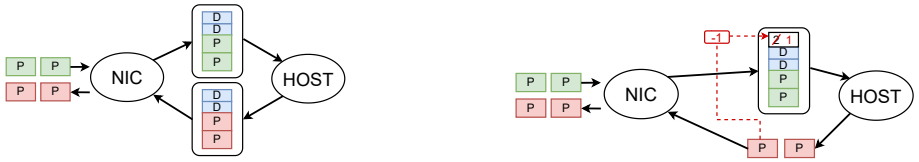
This section describes ASNI's software component on the host. ASNI is currently written in C, but since it reuses the packet interface of standard NICs, it is only a shim layer that could be implemented in other languages. The ASNI library gives a layer of abstraction for packet I/O. This host-side API can be compiled on top of DPDK or X-Change. Recall that X-Change is based on DPDK, and enables a hook to access the hardware ring, still allowing the leveraging of the feature-rich DPDK to set up many NIC offload features, but it is currently restricted to NVIDIA NICs running the MLX5 [8] drivers.

3.1 RX side

The library provides two interfaces to the user, a non-batching version, which abstracts the ASNI frames into a series of packets to be processed one by one by the application, and a batching mode that gives batches of descriptors from the ASNI frame. We show the former in Figure 4. Figure 4a shows how a typical DPDK application receives packets. Figure 4b shows how the ASNI wrapper can be used instead: this is only a simple modification to the packet receive loop. Figure 4c shows the main pseudocode of the receive macro, which demultiplexes packets.

Since the applications might receive multiple ASNI frames, the wrapper is essentially a nested loop. Each ASNI frame starts with a header containing information, such as the number of descriptors. Then the descriptors themselves follow, and the payloads after the descriptors. The *ASNI_end_for_each* macro essentially increments the payload to the packet size.

In the RX path, X-Change avoids the *rte_mbuf* transformation, even for the ASNI frame itself, enabling efficient run-to-completion support. In contrast, DPDK always swaps packet buffers even if the application processes packets immediately and discards them.



(a) Offloaded TX path, the host transmits an ASNI frame

(b) Packetized TX path, reference counting is used to free the ASNI frame

Fig. 5. ASNI forwarding and transmission models

3.2 TX side

ASNI can be used to either offload the RX-side, the TX-side, or both. Forwarding applications can modify individual packets in place in the ASNI frame, and send the Ethernet frame back from the interface it was received. The NIC will then segment the ASNI frame and send individual packets on the wire (Fig. 5a).

Some applications may, however, drop some packets, buffer some packets for reordering, or split batches. The problem is that, despite needing an indirection layer (an array of pointers to the individual descriptors), the application might want to transmit packets from many underlying ASNI buffers, in a different order than they were received. Applications having divergent packet treatments can use standard TX transmission (Fig. 5b). The operation is particularly efficient with the X-Change interface, as the application directly puts the payload's address and length in the hardware TX ring without needing a DPDK descriptor.

Transmission is generally less critical than the RX path, since the burden is put on the NIC to gather packets as for normal packet transmissions. Moreover, recent NICs such as the NVIDIA ConnectX-5 support a function called Enhanced Multi-Packet Write (eMPW) allowing to transmit multiple packet in a single hardware descriptor. Similar to Batchy [29] or Reframer [17], the application can also re-batch packets internally and prepend a segmented list of packets with an ASNI header to pass necessary metadata that would need to be sent to the NIC, enabling ASNI's versatile metadata management for the transmission side.

In any non-forwarding case (indicated by a compile flag given by the application), ASNI initializes a usage counter in the ASNI header. The usage counter is decreased when a packet is dropped or effectively sent. When this counter reaches zero, the ASNI frame (encapsulating all the individual packets) is freed.

4 ASNI SmartNIC Prototype

This section explores our implementation of ASNI on the NIC, including how accelerators, such as the match-action pipeline, can be leveraged to accelerate ASNI. We built a prototype to demonstrate the benefits of enabling different I/O channels with the NIC tailored to each application, focusing on improving the host. Fortunately, today's SoCs-based SmartNICs are sufficiently capable to build prototypes that can sustain >100MPPS rates, only requiring a fraction of the computation capability of the device to realize ASNI's prototype, as illustrated in Section 5.4.

We develop our solution on an NVIDIA BlueField-3 (BF3) SoC [37], a SmartNIC with 16 off-path ARM cores and 32 GB of RAM, running a standard Linux operating system. While our implementation is tied to this architecture, we argue that today's SoC SmartNICs expose similar features [21, 28, 33]. We explore other possible implementations of the ASNI NIC pipeline, for instance, using FPGA SmartNICs or on-path SoC SmartNICs, in Section 7.

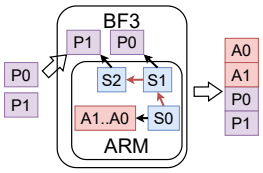


Fig. 7. Reference software-based design

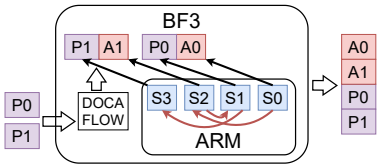
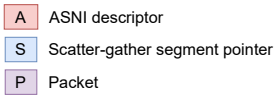


Fig. 8. Match-action pipeline-accelerated solution.



4.1 BlueField 3 SoC

In addition to the ARM CPU cores, the BF3 NIC is equipped with various accelerators (such as crypto, compression, and hash) and ships with libraries [42] to program these accelerators. ASNI relies on specific features of the BF3 platform to enhance its performance. Figure 6 illustrates these features and their interactions with the host. As in traditional NICs, the NIC subsystem handles the physical port and exposes an interface to the host. The whole BF3 is communicating with the host using PCI Express 5.0 x16. The NIC subsystem communicates with the ARM cores using a proprietary bus that exposes itself as PCI Express. The NIC subsystem exposes multiple network interfaces to exchange packets with the ARM cores. One interface is dedicated to traffic from and to the physical port. The second one enables communication with the host. Packet-based communication, therefore, always goes through the NIC subsystem even if one interface is dedicated to ARM-host communication. In addition to kernel drivers, both the ARM CPU (embedded in the BF3) and the host support **DPDK** to transfer packets through the network interfaces.

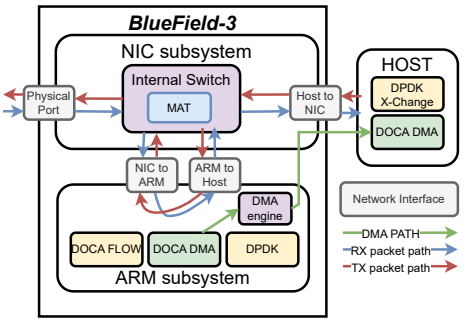


Fig. 6. The BlueField-3 platform

All exposed NIC interfaces support **Scatter-Gather (SG)**, a feature in which the ARM and host CPUs can describe a *single* packet as a list of buffers. The NIC subsystem then gathers the different buffers and sends them as a single packet. The interfaces of the BF3 accept lists of at most 52 buffers. This feature is standard in network interfaces, even in older or more modest hardware such as the Intel 82599 [20]. Packets going through the NIC subsystem go through the **Match Action Tables (MATs)**, enabling the processing of packets in hardware before they reach the ARM and/or the host. Match action tables are configured at runtime, forming a pipeline. The incoming traffic matches a series of rules in each table based on packet fields or metadata. Depending on the matched rules, an action is triggered. The actions include modifying packets, dropping packets, and forwarding packets to another table, a physical port, or a queue. The MAT can modify pre-determined fields, encapsulate the packet, or steer the traffic to specific NIC queues. The tables can be configured either with the DOCA FLOW API [40] or DPDK. The ARM subsystem also embeds a **DMA engine** [38]. The DMA engine enables the ARM to read from and write to HOST memory, but the reverse is not possible.

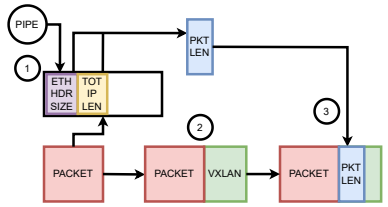


Fig. 9. Retrieving the size through the match-action pipeline

4.2 Reference design

Figure 7 explains our reference software-based implementation. Subsequent sections detail how some accelerators of the BlueField-3 SmartNICs can be used to accelerate the software pipeline.

As detailed in the previous section, the BlueField-3 allows the use of DPDK on the ARM cores to communicate with the HOST. We built the ASNI pipeline with DPDK and compiled it using the header provided by the application (running on the HOST) to construct the ASNI-tailored descriptors. The ASNI pipeline inserts a rule in the MAT to receive all packets intended for the application running on the host.

The pipeline works as follows, as illustrated on Figure 7: (i) The NIC receives a burst of packet (*P0* and *P1*) with DPDK descriptors *S1* and *S2*, respectively. (ii) The ASNI pipeline allocates an extra buffer pointed by DPDK descriptor *S0* that will be used to store all the ASNI (tailored) descriptors. (iii) According to the application requirements, it goes through the packet burst to filter and construct all the ASNI descriptors inside *S0*'s buffer. This results in the aggregated ASNI descriptor sequence *A0* and *A1*. (iv) *S0*, *S1*, and *S2* are chained together and passed to the NIC. Based on this chain, the NIC retrieves the contiguous sequence *A0-A1-P0-P1*. (v) This sequence is transmitted directly to the HOST through ASIC scatter-gather mechanisms, avoiding copying the payloads on the ARM cores. This approach is light on the ARM cores since ASNI only needs to construct a single buffer containing the descriptors and the linked list of packets to be transmitted to the HOST application.

This solution achieves all our desired goals: (i) the application processes continuous chunks of memory; (ii) descriptors are tailored to the application and are therefore minimal; (iii) the overhead of the RX path is significantly reduced on the host. (iv) The construction of descriptors is offloaded to the NIC. (v) We streamline the interaction between the NIC and the HOST. Instead of requiring one PCIe read per packet descriptor, the NIC only needs one per batch, saving PCIe bandwidth and reducing the number of necessary transactions. We analyze these gains in Section 5.

4.3 Acceleration using the match-action pipeline

The BlueField-3 provides Match-Action Tables (MATs) to process packets before they reach the ARM cores. We propose using these MATs to construct tailored descriptors directly in hardware, thereby lessening the load on the ARM cores, as shown in Figure 8. The MATs' rules encapsulate the packet with a new header (such as VXLAN) to push metadata (such as RSS hash and packet length) and data from the packet itself (IP address, ToS field, etc.). The Match-Action Tables offer a 32-byte shared metadata region (SMR) that is accessible by different pipelines. In addition to transferring data between the SMR and the packets, pipelines can perform simple computations, such as adding values from the packet to the SMR. For example, retrieving the packet size can be accomplished entirely in hardware for IP packets, as illustrated in Figure 9. (1) First, we copy the size of the Ethernet header and the total IP length into the SMR. (2) The packet is encapsulated into

a VXLAN header. (3) The NIC computes the total size of the packet from the SMR. The result is copied into the VXLAN header, which now acts as metadata for the packet.

Once received on the NIC, the batch of descriptors can be chained on the ARM core and then forwarded without transforming them into software. As shown in Figure 8, we configure the NIC to split each incoming packet into two separate buffers: the hardware-constructed (encapsulation) header and the actual packet. Splitting incoming packets into separate buffers is a standard feature exposed by DPDK. The BlueField 3 MATs allow only some encapsulation types, limiting the size of descriptors to the size of the supported encapsulations, and only supports a fixed amount of fields and protocols [40].

4.4 DMA-based implementation

Our initial design received packets on the NIC with DPDK and transmitted transformed descriptors with DMA directly to the host memory. Previous work [57] showed that the performance of the DMA API of the BlueField-2 is low for small transactions. Although the BlueField-3 slightly improves performance, the SoC's DMA engine is still much weaker than the ASIC packet engine used by the software implementation. We therefore discarded the DMA-based implementations. We describe this DMA-based implementation and its underperformance compared to the other designs in Appendix B.

4.5 Multiple pipelines and multiple applications

Piggybacking the Ethernet API enables an easy and flexible deployment of our solution. In particular, we can rely on well-supported NIC virtualization technologies to deploy multiple applications, as shown in Figure 10. As DPDK applications bypass the kernel, they cannot share the same physical NIC. SR-IOV [41] is a technology that enables exposing a NIC as multiple PCIe devices called Virtual Functions (VF). SR-IOV is primarily intended for virtualization, enabling dedicated virtual functions for different guests. Each VF acts as a virtual NIC, supporting multiple queues within each VF. Therefore, using SR-IOV, each application has its own (virtual) dedicated NIC. When SR-IOV is used on the HOST, a Virtual NIC port corresponding to the VF is also exposed to the ARM cores. If a packet is sent on virtual port number 2 on the ARM cores, it will be received by the host's second virtual function. As a result, we only need to run parallel ASNI pipelines on the ARM cores, one per application, which will receive traffic dedicated to the correct application, build the ASNI frames, and deliver them to the corresponding virtual port on the ARM cores. Each ASNI pipeline will receive the correct portion of traffic using hardware steering (e.g., traffic directed to TCP port 80). The host application can then attach to the right SR-IOV VF and receive the ASNI frames made by its own tailored pipeline.

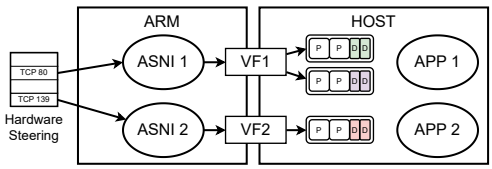


Fig. 10. Two processes running with ASNI.

We also have the challenge of sharing the ARM cores between the multiple ASNI pipeline instances. In our testing, we limited our study to static core allocations: each ASNI pipeline serving an application on the HOST is given a fixed set of ARM cores. Future work could dynamically allocate more or fewer resources to the NIC based on the pipeline's current load.

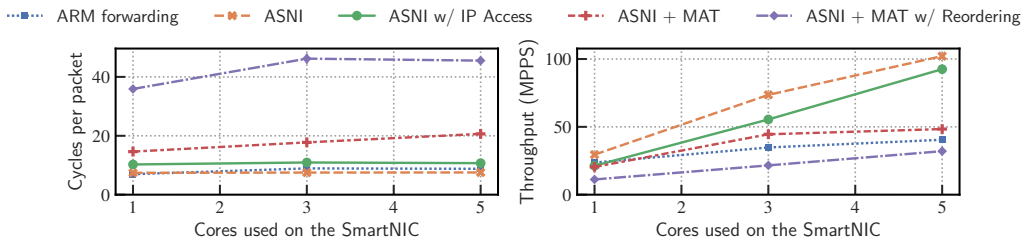


Fig. 11. Cycles cost (the internal TSC clock frequency is 200 MHz on the BlueField-3) and throughput according to the number of cores of the NIC

5 Evaluation

In this section, we first evaluate the performance of our reference implementation. We analyze the impact of the accelerators enabled by the BlueField-3 SmartNIC and assess ASNI's improvement for applications running on the host. We answer the following questions: (i) What are the impacts on the performance of the accelerators of the BlueField-3 SmartNIC in Section 5.2; (ii) Which layout for descriptors and payloads is most suitable for the host, in Section 5.3 and 5.4; (iii) How does batching impact the performance of ASNI, in Section 5.5; (iv) How does the size of the descriptor impact performance in Section 5.6; (v) Does ASNI provides improvement for real applications running on the host in Section 5.7. We focused our evaluation on NFV applications and a Key/Value Store. We don't expect improvements for applications where the bottleneck is unrelated to I/O.

5.1 Testbed

To generate traffic at a high rate, we rely on P4TG [31], a traffic generator based on P4 Tofino switches, which can output packets at a line rate with 64 B packets, equivalent to around 147 MPPS. In addition to generating packets at high speed, P4TG can also provide precise measurements of latency. We use synthetic traffic at fixed rates and homogeneous packet sizes for all our measurements.

An Ubuntu 22.04 machine equipped with an Intel Xeon Silver 4314 CPU, running at 2.4 GHz with 16 cores, executes the test applications using a BlueField-3 200G NIC. It is, however, limited by its 100 Gbps cable, which is connected to a 100 Gbps port on the generator switch. To showcase the improvement on the CPU, we use only one CPU core on the host. We set the CPU frequency to its nominal 2.4 GHz to obtain more precise measurements and reduce variance, especially when considering the cycles required for processing. We use a similar machine equipped with a ConnectX-6 100 Gbps NIC for the NAT experiment shown in Section 5.7.1 to be able to generate L4 traffic using Pktgen [58] and for Section 5.7.3 to run the mica2 request generator. We use DPDK 23.11 on the host and the provided DPDK version with DOCA 2.7 on the SmartNIC. The original X-Change [11] is not compatible with our BlueField-3 as it is based on an older version of DPDK. We ported the RX path of X-Change to DPDK 23.11. Since the transmit path is less critical for application performance, we leave it for future work. As such, applications that forward packets use ASNI over DPDK, not the X-Change wrapper. The ASNI reference implementation on the SmartNIC fits inside ~ 1000 Lines of Code (LoC). The host library is ~ 800 LoC. The other variants are ~ 2000 LoC.

5.2 SmartNIC pipeline evaluation

In this section, we evaluate the ASNI SmartNIC prototype and its accelerations described in Section 4. Figure 11 shows the number of cycles per packet needed for processing on the ARM cores and the goodput of the SmartNIC implementation in Million Packets Per Second (MPPS). The baseline (ARM

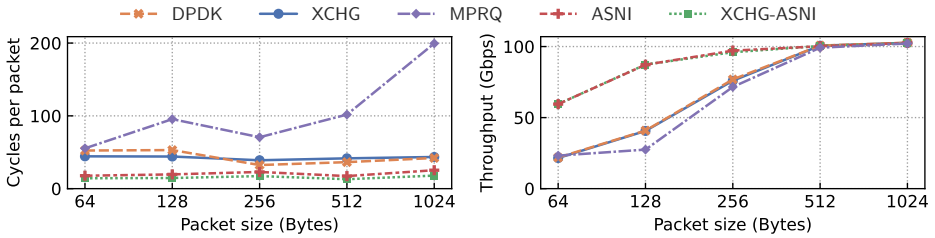


Fig. 12. Number of cycles on the host and throughput according to the data path framework

forwarding) is a DPDK application that forwards packets on the SmartNIC. In this experiment, the client sends 64-byte packets to stress the I/O subsystem and is not limited by the line rate. These packets are crafted with ranges of IP addresses and ports to distribute them evenly across the NIC's Receive Side Scaling (RSS) queues. In total, we generated 16,000 distinct flows, which we found to be sufficient for our testing purposes. On the left, we observe the cost in cycles per packet of each application. ASNI outperforms the forwarding application running on the ARM cores even when limited to a single core on the SmartNIC. *ASNI w/ IP Access* showcases the cost of accessing the payload (the source IP) to place it inside the tailored descriptor, resulting in a 12 % throughput reduction when running on five cores on the SmartNIC.

We evaluate the ASNI MAT-accelerated solution. The *ASNI + MAT* line outlines the performance of ASNI using the Match-Action Table to accelerate the pipeline. The results show that MAT itself does not allow descriptors to be batched together; therefore, reordering is required, as the *ASNI + MAT* line does not follow the ASNI frame layout. We leverage the *rx_split* [6] functionality of the NIC, which can arbitrarily split packets received from the MAT into multiple buffers based on an offset within the packet. This functionality is standard in DPDK and is implemented in the MLX5 drivers. In our case, we receive the MAT-constructed descriptor in one buffer and the rest of the packet in another. The buffers can then be reordered before transmission to the host using scatter/gather, similar to the reference design in Section 4.2. Overall, the MAT limited performance when the number of tables and thus the complexity of the pipeline increases, already shown by previous work [24] prevents this acceleration from being effective. **For the rest of the paper, we use the ASNI reference software packet-based solution without any of the acceleration mechanisms as this is the version that gives us the best performance.**

5.3 ASNI Data Paths

As a microbenchmark, we use Flowatcher [59], an application that collects per-flow statistics. After processing, the packets are dropped, as we only evaluate the RX path. We first evaluate the effectiveness of ASNI against DPDK and XCHG forwarding packets. We modify FloWatcher to act as a simple packet counter, where only the packet size is required, and no system accesses the payload. Figure 12 shows the per-packet cost for the different data paths, as well as the achieved throughput.

NVIDIA ConnectX-5 and above, provide the Multi-Packet Receive Queue (MPRQ) feature, enabling the use of a single buffer to receive multiple packets. Similarly to ASNI, this feature releases PCIe bandwidth and can improve RX performance, particularly for small packets. However, parameter tuning for each packet size is required for maximum performance. In our experimental setup, we tuned the values for 64-byte packets, which gives a slight performance boost only for this particular size. MPRQ still requires the construction of per-packet descriptors. Figure 12 highlights that, while it reduces the load on the PCIe, it is not effective at improving throughput or packet processing times; its cost even rises when the packet size increases.

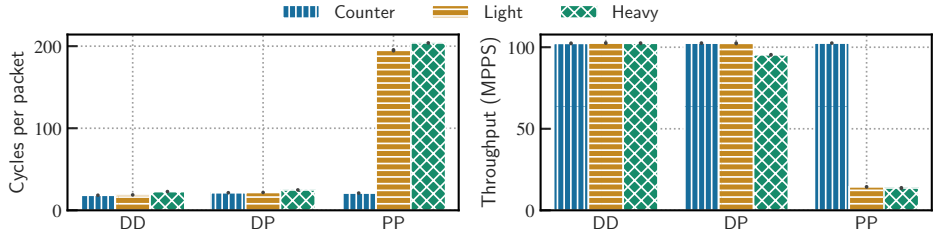


Fig. 13. Impact of the frame layout: batched descriptors (DD), interleaved (DP), and payloads only (PP)

Regardless of the packet size, the two ASNI versions outperform DPDK, XCHG, and MPRQ. ASNI reduces the per-packet cost by 3.1×. For this experiment, XCHG outperforms DPDK in terms of cycles per packet at low packet sizes. However, the difference does not translate to better throughput, which suggests that the bottleneck is the NIC. DPDK and XCHG outperform MPRQ for packet sizes different from 64 since the parameters are no longer adequate. When comparing cycle costs between ASNI and XCHG-ASNI, we see the same phenomenon as between XCHG and DPDK.

5.4 Format of the ASNI frame

We next evaluate the order of descriptors and payloads and their impact on performance.

We evaluate 3 variation of our micro-benchmark application, each with different memory access patterns and computations. We use the packet counter from the previous section (labeled Base). A light workload that adds the hashes of every received packet (labeled Light), and the normal FloWatcher application, using the RSS hash to count flows in a memory table (labeled Heavy).

We evaluate ASNI with 3 different descriptors and packet layouts. (i) PP (Payload-Payload) : the ASNI frame comprises contiguous payloads, without descriptors. (ii) DP (Descriptor-Payload) : the host receives a continuous array of interleaved payloads and descriptors. (iii) DD (Descriptor-Descriptor) : all the descriptors are grouped together followed by all the payloads, as in the reference ASNI design.

Figure 13 shows that the DD and DP versions have comparable performance except in the case of heavy processing, where the better cache locality of the descriptors in the DD variant is beneficial. When a hash of the packet is necessary for the PP version, it is computed in software using DPDK built-in functions.

The observed throughput in Figures 13 are closely correlated with the per-packet costs. For all scenarios, the DD version handles more than 100 MPPS on a single CPU core.

Furthermore, we observe that at high speeds, even standard processing such as computing a hash on the packet headers can significantly impact the performance: for the light and heavy workloads *PP* offers 5x less throughput. The software hash is computed using the built-in DPDK Hash Library [7], which can compute a hash similar to an RSS hash calculated by the hardware. The PP version is comparable to Ensō since the host receives batches of multiple contiguous payloads as a single transfer (there are no descriptors) and a single notification through the standard NIC ring for all those batched payloads. Ensō very similarly sends a contiguous payload in a unique buffer and a notification in another channel. We thus expect Ensō to struggle with similar workloads. With the advent of more advanced NICs, we expect an increase in computation handled by the NICs itself. We conclude from this experiment that using descriptors is essential to transmit metadata to the CPU.

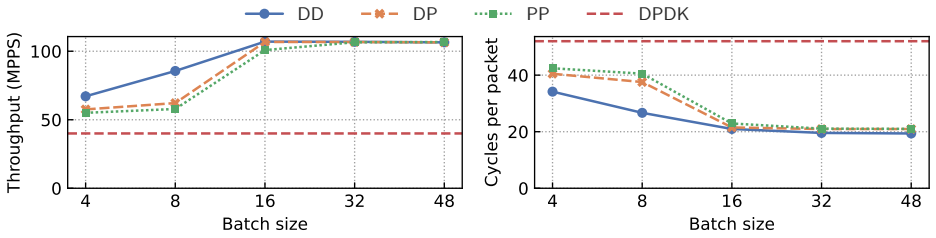


Fig. 14. Observed throughput and cycles consumption depending on frame format for FloWatcher

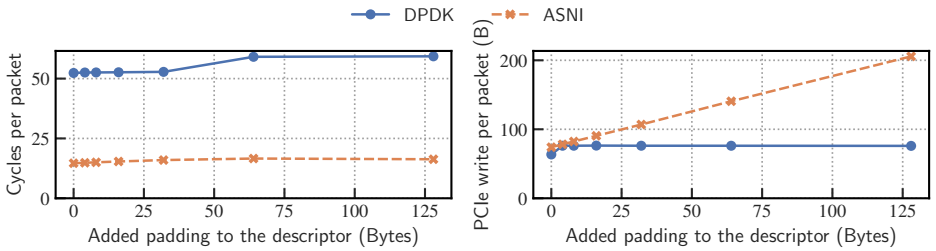


Fig. 15. Impact on PCIe consumption and cycles per packet cost depending on descriptor size

5.5 ASNI frame size

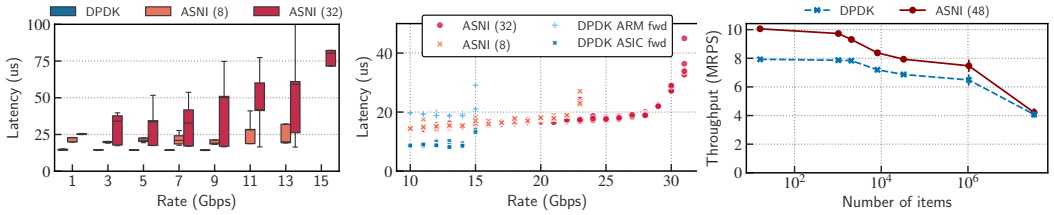
Using jumbo frames to transmit batches of packets from the HOST to the NIC imposes a limitation on the size of batches transmitted. On the BF3, the maximum achievable MTU is 9978B, meaning that we can transmit at most 9 1024B packets and their descriptors inside a single jumbo frame. When using smaller packets, we can go up to 51 packet inside a single ASNI frame, since 52 is the maximum size of a scatter-gather list on the BF3, and we need one extra node for the metadata.

Figure 14 shows the obtained throughput with varying ASNI frame sizes. Even small frames of 4 or 8 packets ASNI can outperform DPDK, which cannot exceed 40MPPS. The differences between the other ASNI implementations are also highlighted in this scenario. The better performance of packed descriptors (DD) can be explained by better cache locality for the descriptors. For instance, with 8-byte descriptors, the descriptors can all fit in a single cache line. In contrast, to retrieve all the sizes, the other implementations have to go through the whole jumbo frame, which significantly reduces performance.

5.6 ASNI descriptor size

We evaluate the impact of using larger application descriptors. Figure 15 shows the effect on the number of cycles required for processing on the host compared with DPDK and ASNI depending on the descriptor size. DPDK has 36 contiguous bytes available in its application descriptor. After that, we use a DPDK mempool to allocate more space and 8 bytes to store a pointer to the dynamically allocated space.

We note that with small supplementary padding, ASNI's batching compensates for the 8 bytes used for individual packets and 16 bytes of the ASNI header. As the descriptor becomes bigger, the number of cycles on the host does not increase much, although the PCIe bus sees an overhead that may become a bottleneck depending on the available spare bandwidth.



(a) Vignat latency as a function of the throughput for ASNI (batching 8 or 32 packets) and DPDK

(b) Latency as a function of the throughput for Maglev

(c) Throughput in Millions Packets Per Second for MICA2 with DPDK and ASNI

Fig. 16. Three realistic applications. ASNI improves the throughput with a negligible latency cost.

5.7 Realistic applications

We finally evaluate ASNI on 3 realistic applications: a NAT, a load-balancer, and a Key-Value Store. All applications use ASNI over DPDK, as we have not yet ported the TX side of X-Change to a version of DPDK compatible with the BlueField-3.

5.7.1 NAT: Vignat. We use Vignat [48] as our NAT use case. It is slightly improved with standard high-speed techniques to avoid I/O bottleneck (more details in appendix A.1). We generate traffic with 64-byte packets and from 1 to 15 Gbps transmission rates to evaluate the NAT. We report the obtained latencies with varying throughput. Only points where the loss rate is below 1% are considered. We use use a minimal ASNI descriptor that only contains the size of the packet. We observe in Figure 16a that ASNI manages to serve 66% more traffic than a pure DPDK-based implementation. The latency is reported for ASNI configured to batch at maximum 8 or 32 packets in an ASNI frame. Batching more packets slightly improves performance, but adds latency and more variance even for low throughputs. For example, starting from 3 Gbps, batches of 32 add 20 us of latency compared to batches of 8.

5.7.2 Load-balancer: Maglev. We choose the Maglev [9] load-balancer implementation of Ensō [52] which uses DPDK. The ASNI descriptor used only contains the size of the packet, similarly to Vignat. We can see that ASNI manages to serve 2.2X more packets compared to DPDK. To evaluate the latency induced by ASNI itself, we evaluate a scenario where we run the DPDK variant on the host without ASNI. The NIC either forwards the packet using hardware rules or using a DPDK program that forwards packets in software. The results are in Figure 16b. We see that merely using DPDK to forward packets on the NIC adds 10us to the round-trip time. When using ASNI, this deficit shrinks to 5us, which can be explained by the fact that the ARM and the HOST receive and send fewer individual packets overall as they exchange ASNI frames instead, providing a lower latency. The key takeaway is that in this scenario, ASNI does not add any extra latency compared to ARM forwarding, highlighting how light the processing is on the ARM cores. We can also observe that batching here again significantly improves the performance. By moving from batches of 8 packets to 32 packets, we go from 22Gbps to 30Gbps before the latency increases sharply due to increased queuing. At the host level, the processing is much more efficient, as shown by the observed throughput.

5.7.3 Key/Value Store: mica2. Mica2 [30] is a high-performance key-value store that leverages Data Plane Development Kit (DPDK) for its operations. This application's ASNI descriptor is more complex since we need to buffer incoming requests and craft packets containing the responses. In

addition to storing the packet size, the ASNI descriptor includes pre-allocated space for a pointer to the packet, along with metadata about the large buffer, used for freeing the memory once all packets in the ASNI frame have been processed. Mica2 requires buffering incoming packets and generating new response packets. In such applications, I/O operations are not necessarily the primary bottleneck; rather, the application's processing consumes most CPU cycles. Since we are only offloading the RX path, the modifications required to run the applications are minimal.

Figure 16c shows the throughput obtained with different database sizes for Mica2. We set the ratio of get/set to 50%. We can see that, with a low number of elements, ASNI offers significant improvements compared to DPDK. With a million entries, the processing of the application itself consumes most of the CPU cycles. This example shows that ASNI also improves performance for applications with request/response patterns.

6 Related works

Ensō [52] proposes a complete redesign of the network interface, enabling it to achieve 147 MPPS with a single CPU core. ASNI reaches 110 MPPS in the same scenario. Beyond this point, the NIC ASIC becomes the bottleneck as the ARM cores cannot forward more packets. Ensō comes at the cost of removing metadata support and sacrificing zero-copy. We could not compare directly ASNI to Ensō as it uses a $10K FPGA, which we do not have at our disposal. However, we believe Ensō could benefit from receiving ASNI-like tailored descriptors to pass some metadata and avoid the 5x performance collapse shown in Section 5.4 when the application must recompute a piece of information readily available in hardware. On top of the other models described in Section 2 (PacketMill [11]'s X-Change, and TinyNF [47]), proposing a different design between the host and the NIC, there are a few related works worth mentioning.

IDPF [44] is a new initiative which proposes the opposite of what we want to achieve: have a universal driver that has the software adapt to the NIC. This eases driver maintenance but burdens even more the software datapath. FlexNIC [26] is a proposal for a DMA API for NICs. It proposes to install packet rules on the NIC, allowing it to send only the interesting portion of the packets to the host. NICBench [24] characterizes the classification performance of SmartNICs. Both, however, do not focus on the buffering model. FlexNIC only proposes an emulated implementation. A contribution of this paper is the evaluation of an actual design to implement ASNI on a SmartNIC.

Lynx [54] proposes to offload some part of the network stack to accelerators and customize the interface between NIC and software, targeting GPU applications. Similarly, there has been a lot of research to offload TCP stacks [4, 53] on FPGAs or SoC SmartNIC [32, 35]. NICA [10] proposes an abstraction to ease the acceleration of application on FPGA NICs. Authors enable a "custom ring" to directly provide software with UDP payloads without the need for the host to process headers. In this work, we focus on the means of communication, and particularly how the host should receive and transmit data. It is orthogonal to offloading.

Floem [46] proposed a more fluid split of duties between the CPU and the NIC and gives a framework for developers to explore different offload designs. However, our works are orthogonal. We investigate how the transfer between the NIC and the host should be tailored. As Floem pushes more offloads to the NIC, more metadata would need to be passed with the packet. In addition, computing units and accelerators on the NIC itself could take advantage of ASNI and work on large continuous segments. Therefore, ASNI would be beneficial to Floem. NanoPU [19] proposes to receives the most important part of the packet directly in a RISC V CPU registers. Using this technique, ASNI would allow multiple descriptors to be sent at once directly in those registers.

7 Discussion

We used the BF3 platform to demonstrate the benefits for the host. In this section, we discuss the generality of our proposal, mainly whether ASNI could be implemented using other NICs.

Could ASNI be implemented using other SoC-based SmartNICs? Some SmartNICs embed on-path accelerators [34, 36, 45]. On-path cores would enable lower latency when transforming descriptors to the ASNI-tailored ones. None of these alternatives offers a complete public datasheet. To avoid payload copy on the cores themselves, we used the scatter-gather interface, which is generally available in NIC interfaces, to indicate that packet buffers are to be assembled as a unique packet.

The BlueField-3 also embeds a RISC-V CPU with 16 cores and 256 threads called a DPA [39]. We believe that ASNI could be implemented on this platform as it is a general-purpose CPU and can also use a scatter-gather API to avoid copying the payload. Previous work [5] has shown that it cannot manage as much traffic in packet processing workloads as the ARM CPU, although it provides a lower latency with lower loads as it is closer to the NIC than the BF3 off-path ARM cores, thus ASNI on the DPA would provide less throughput with a possible gain in latency.

Could P4-enabled SmartNICs be useful to ASNI? Descriptors could be built entirely with a proper P4 pipeline, as we show with the MAT acceleration in Section 4.3. In ASNI, descriptors are defined using a set of C flags, allowing individual fields to be enabled selectively. However, this approach does not support reordering fields or expressing data transformations. To enable an even more tailored descriptor, the P4 language could be used to describe the metadata field pushed in front of a packet. NVIDIA recently released a P4 compiler [43] for the same hardware MAT pipeline we use in this paper. This compiler should therefore provide the same performance, but enable a more flexible definition of the descriptor. As shown in Section 5.2, the MAT acceleration is limited by the performance of the pipeline to 45MPPS, while the ARM-based version can reach 110MPPS. Some SmartNICs [21, 45] embed a dedicated P4 pipeline which might offer a higher throughput.

Could ASNI be implemented on FPGAs SmartNICs? ASNI is not limited to SoC-based SmartNIC implementations. We used the BF3 as a convenient prototyping platform. P4FPGA [56] demonstrated the ability to use a MAT pipeline at line rate, while scatter-gather functionalities have also been implemented in NetFPGA [1].

Should ASNI be implemented in ASIC? ASNI provides significant gains for the host. Therefore, we believe it should eventually be implemented in the NIC ASIC. While a MAT pipeline allows for the creation of custom descriptors, there is currently no standard way to define how descriptors and payloads should be batched together, as we do in ASNI. This task is computationally similar to Large Receive Offload (LRO), where the payloads of multiple TCP packets are packed together, and a newly constructed header is added in front of the reconstructed payload. Therefore, we speculate that the circuitry is already available.

8 Conclusion

We presented ASNI, a new packet management model. ASNI uses a SmartNIC to prepare packets according to the application's design choice. ASNI improves host packet processing performance in two ways. First, ASNI batches both descriptors and payloads into a single continuous buffer, limiting the number of PCIe transactions and improving cache locality for both payloads and descriptors. Still, ASNI maintains the ability to forward and buffer packets with zero-copy. Second, ASNI is using smaller and tailored descriptors. This reduces the application's memory footprint, especially by packing multiple descriptors in a single cache line.

In NFV scenarios, we manage to serve up to 2.2x more traffic. Our solution also showed to be relevant for applications with a producer/consumer pattern, such as key-value stores. One complexity

of ASNI's implementation lies in finding the right implementation to accommodate descriptors on the SmartNIC. We showcased different accelerators available to the NIC and highlighted their strengths and weaknesses. We showed that borrowing the existing rich Ethernet API is a good way to manage the exchange of buffers between the SmartNIC and the host's application.

In addition to CPUs, the use of accelerators or coprocessors is becoming more common to sustain the ever-increasing network speeds. We believe our new interface could be especially suitable for accelerators such as GPUs that can work more efficiently on continuous structures, receiving the packed descriptors as a vector ready to be processed in parallel chunks. Moreover, implementing ASNI in ASIC could enhance the capabilities of processing cores on the NIC itself, enabling even lower-power SmartNIC cores to handle higher loads. In future work, we would like to bridge the gap between our proposal of an interface between (smart) NICs and the application, and performance models for offloading. Tools such as Clara [49] and performance interface [22] would allow considering multiple possible ways to offload (or not) a feature that should, one way or another, be given in the packet descriptor. ASNI is available at https://github.com/UCLouvain-ENSG/ASNI.

Acknowledgments

The authors would like to acknowledge the anonymous reviewers for their valuable feedback. We want to thank our Shepherd, Akshay Narayan, for his constructive feedback and careful review of our work. We also thank NVIDIA for granting a BlueField-2 used in prototyping and Intel for granting the Tofino used as a packet generator. This project has been funded by the Fonds National de la Recherche Scientifique - FNRS through the MIS grant 40020886 and UCLouvain FSR. Clément Delzotti is a beneficiary of the FNRS FRIA grant 40022220. This research is supported by the Walloon region's CyberExcellence program (grant #2110186).

References

[1] Gianni Antichi, Christian Callegari, and Stefano Giordano. 2013. Implementation of TCP large receive offload on open hardware platform. In *Proceedings of the first edition workshop on High performance and programmable networking (HPPN '13)*. Association for Computing Machinery, New York, NY, USA, 15–22.

[2] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. 2019. RSS++: load and state-aware receive side scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies (CoNEXT '19)*. ACM, Orlando Florida, 318–333. doi:10.1145/3359989.3365412

[3] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast userspace packet processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (Oakland, California, USA) *(ANCS '15)*. IEEE Computer Society, Washington, DC, USA, 5–16. http://dl.acm.org/citation.cfm?id=2772722.2772727

[4] Junehyuk Boo, Yujin Chung, Eunjin Baek, Seongmin Na, Changsu Kim, and Jangwoo Kim. 2023. F4T: A Fast and Flexible FPGA-based Full-stack TCP Acceleration Framework. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*. Association for Computing Machinery, New York, NY, USA, 1–13.

[5] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, et al. 2024. Demystifying datapath accelerator enhanced off-path smartnic. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP '24)*. IEEE, IEEE, New York, NY, USA, 1–12.

[6] DPDK. 2025. *API documentation - RX Split*. Linux Foundation. https://doc.dpdk.org/api/structrte__eth__rxseg__split.html Accessed on 16th April, 2025.

[7] DPDK. 2025. *DPDK Hash Library*. Linux Foundation. https://doc.dpdk.org/api/rte__thash_8h.html Accessed on 25th April, 2025.

[8] DPDK. 2025. *DPDK MLX5 drivers*. DPDK. https://doc.dpdk.org/guides/nics/mlx5.html Accessed on 25th April, 2025.

[9] Danielle E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*. USENIX Association, Santa Clara, CA, 523–535. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud

[10] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. 2019. NICA: An infrastructure for inline acceleration of network applications. In *2019 USENIX Annual Technical Conference (ATC '19)*. USENIX Association, Renton, WA, 345–362.

[11] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2021. PacketMill: toward per-Core 100-Gbps networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Virtual USA, 1–17. doi:10.1145/3445814.3446724

[12] fd.io. 2023. Vector Packet Processing (VPP). https://fd.io/technology/

[13] FD.io. 2023. *VPP: Direct support for mlx5_dv*. FD.io. https://gerrit.fd.io/r/c/vpp/+/24240 Accessed on 9th August, 2023.

[14] Linux Foundation. 2023. *DPDK: Data Plane Development Kit*. Linux Foundation. https://www.dpdk.org/ Accessed on 28th November, 2023.

[15] Linux Foundation. 2024. *DPDK API documentation - mbuf*. Linux Foundation. https://doc.dpdk.org/api/structrte_mbuf.html

[16] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, Renton, WA, 281–297.

[17] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan Kostić. 2022. Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)*. USENIX Association, Renton, WA, 807–827.

[18] Jack Tigar Humphries, Neel Natu, Kostis Kaffes, Stanko Novaković, Paul Turner, Hank Levy, David Culler, and Christos Kozyrakis. 2024. Tide: A Split OS Architecture for Control Plane Offloading. *arXiv preprint arXiv:2408.17351* (2024).

[19] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. 2021. The nanopu: A nanosecond network stack for datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)*. USENIX Association, Boston, MA, 239–256.

[20] Intel. 2024. *datasheet*. Intel. https://cdrdv2-public.intel.com/331521/331521_82599_SpecUpdate_Rev4.3.pdf

[21] Intel. 2025. *Intel® Infrastructure Processing Unit*. Intel. https://www.intel.com/content/www/us/en/products/details/network-io/ipu/e2000-asic.html

[22] Rishabh Iyer, Jiacheng Ma, Katerina Argyraki, George Candea, and Sylvia Ratnasamy. 2023. The Case for Performance Interfaces for Hardware Accelerators. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems* (Providence, RI, USA) *(HOTOS '23)*. Association for Computing Machinery, New York, NY, USA, 38–45. doi:10.1145/3593856.3595904

[23] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. 2012. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12)*. ACM, New York, NY, USA, Article 9, 14 pages. doi:10.1145/2391229.2391238

[24] Georgios P Katsikas, Tom Barbette, Marco Chiesa, Dejan Kostić, and Gerald Q Maguire Jr. 2021. What You Need to Know About (Smart) Network Interface Cards. In *Passive and Active Measurement (PAM '21)*. Springer International Publishing, Cham, 319–336.

[25] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. 2018. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 171–186. https://www.usenix.org/conference/nsdi18/presentation/katsikas

[26] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM Press, Atlanta, Georgia, USA, 67–81. doi:10.1145/2872362.2872367

[27] Joongi Kim, Seonggu Huh, Keon Jang, KyoungSoo Park, and Sue Moon. 2012. The power of batching in the Click modular router. In *Proceedings of the Asia-Pacific Workshop on Systems* (Seoul, Republic of Korea) *(APSYS '12)*. Association for Computing Machinery, New York, NY, USA, Article 14, 6 pages. doi:10.1145/2349896.2349910

[28] Xsight Labs. 2025. *Xsight Labs E1*. Xsight labs. https://xsightlabs.com/wp-content/uploads/2024/10/Xsight-Softens-the-DPU.pdf

[29] Tamás Lévai, Felicián Németh, Barath Raghavan, and Gabor Retvari. 2020. Batchy: Batch-scheduling Data Flow Graphs with Service-level Objectives. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*. USENIX Association, Santa Clara, CA, 633–649.

[30] Hyeontaek Lim, Donsu Han, David G Andersen, and Michael Kaminsky. 2014. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*. USENIX Association, Seattle, WA, 429–444.

[31] Steffen Lindner, Marco Häberle, and Michael Menth. 2023. P4tg: 1 tb/s traffic generation for ethernet/ip networks. *IEEE Access* 11 (2023), 17525–17535.

[32] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading distributed applications onto smartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication*. Association for Computing Machinery, New York, NY, USA, 318–333. doi:10.1145/3341302.3342079

[33] Marvell. 2025. *Marvel Octeon 10 DPU.* Marvell. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-octeon-10-dpu-platform-product-brief.pdf

[34] Marvell. 2025. *Marvell liquidio iii.* Marvell. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf,2022

[35] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 77–92.

[36] Netronome. 2025. *Agilio CX.* Netronome. https://netronome.com/wp-content/uploads/PB_Agilio_CX_2x10GbE-3-24.pdf

[37] NVIDIA. 2023. BlueField Data Processing Units (DPUs). https://www.nvidia.com/en-us/networking/products/data-processing-unit/

[38] NVIDIA. 2023. *DOCA DMA.* NVIDIA. https://docs.nvidia.com/doca/sdk/dma-programming-guide/index.html Accessed on 29th November, 2023.

[39] NVIDIA. 2023. *DOCA DPA.* NVIDIA. https://docs.nvidia.com/doca/sdk/dpa-subsystem-programming-guide/index.html Accessed on 29th November, 2023.

[40] NVIDIA. 2024. *DOCA FLOW.* NVIDIA. https://docs.nvidia.com/doca/archive/2-7-0/doca+flow/index.html Accessed on 20th April, 2025.

[41] NVIDIA. 2025. *BlueField SR-IOV.* NVIDIA. https://docs.nvidia.com/networking/display/bluefielddpuosv470/bluefield+sr-iov Accessed on 16th April, 2025.

[42] NVIDIA. 2025. DOCA API. https://docs.nvidia.com/doca/sdk/doca+libraries/ Accessed on 30th April, 2025.

[43] NVIDIA. 2025. *DPL.* NVIDIA. https://docs.nvidia.com/doca/sdk/dpl+system+overview/index.html Accessed on 16th April, 2025.

[44] OASIS Open. 2023. *IDPF.* OASIS Open. https://www.oasis-open.org/2023/01/11/creating-a-leading-peripheral-component-interconnect-express-pcie-based-ethernet-host-interface-standard-infrastructure-data-plane-function-idpf/ Accessed on 2nd December, 2023.

[45] AMD Pensando. 2025. *Pensando DPU.* AMD. https://www.amd.com/en/products/accelerators/pensando.html

[46] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A programming system for NIC-Accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. USENIX Association, California, United States, 663–679.

[47] Solal Pirelli and George Candea. 2020. A Simpler and Faster NIC Driver Model for Network Functions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Renton, WA, 225–241.

[48] Solal Pirelli, Arseniy Zaostrovnykh, and George Candea. 2018. A Formally Verified NAT Stack. In *Proceedings of the 2018 Afternoon Workshop on Kernel Bypassing Networks* (Budapest, Hungary) *(KBNets '18)*. Association for Computing Machinery, New York, NY, USA, 8–14. doi:10.1145/3229538.3229540

[49] Yiming Qiu, Qiao Kang, Ming Liu, and Ang Chen. 2020. Clara: Performance Clarity for SmartNIC Offloading. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks* (Virtual Event, USA) *(HotNets '20)*. Association for Computing Machinery, New York, NY, USA, 16–22. doi:10.1145/3422604.3425929

[50] Luigi Rizzo, Giuseppe Lettieri, and Vincenzo Maffione. 2013. Speeding up packet I/O in virtual machines. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (San Jose, California, USA) *(ANCS '13)*. IEEE Press, Piscataway, NJ, USA, 47–58.

[51] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. 2019. Elastic RSS: Co-Scheduling Packets and Cores Using Programmable NICs. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking (APNet '2019)*. ACM, Beijing China, 71–77. doi:10.1145/3343180.3343184

[52] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S. Berger, James C. Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. 2023. Enso: A Streaming Interface for NIC-Application Communication. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23)*. USENIX Association, Boston, MA, 1005–1025. https://www.usenix.org/conference/osdi23/presentation/sadok

[53] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. 2015. Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware. In *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM '15)*. IEEE Computer Society, USA, 36–43. doi:10.1109/FCCM.2015.12

[54] Maroun Tork, Lina Maudlej, and Mark Silberstein. 2020. Lynx: A SmartNIC-driven Accelerator-centric Architecture for Network Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 117–131. doi:10.1145/3373376.3378528

[55] Björn Töpel. 2018. Introducing AF_XDP support. https://lwn.net/Articles/745934/

[56] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. ACM, New York, NY, USA, 122–135.
[57] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. Characterizing Off-path SmartNIC for Accelerating Distributed Systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23)*. USENIX Association, Boston, MA, 987–1004.
[58] Keith Wiles. 2023. Pktgen - Traffic Generator powered by DPDK. https://github.com/pktgen/Pktgen-DPDK
[59] Tianzhu Zhang, Leonardo Linguaglossa, Massimo Gallo, Paolo Giaccone, and Dario Rossi. 2019. FloWatcher-DPDK: Lightweight line-rate flow-level monitoring in software. *IEEE Transactions on Network and Service Management* 16, 3 (2019), 1143–1156.

A Notes on applications

A.1 Vignat modifications

Vignat [48] is a NAT designed for formal verification. Since we don't target formal verification software, we modified the implementation slightly to achieve higher performance and shift the bottleneck towards IO without changing its core behaviour. Changes done include (1) Moving the checksum calculation to the hardware (2) Using more efficient time retrieval using the Time Stamp Counter (TSC) (3) Changing the IO functions to run on top of our API.

B DMA-based implementation

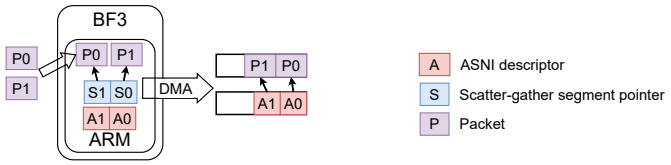


Fig. 17. DMA-accelerated version (the receive format deviates from the reference design)

We implemented a variant of ASNI using Direct Memory Access (DMA) [38] and DOCA 2.5 to write descriptors in host memory, avoiding the packetized interface. On the host side, no code change is required to use this mode with the ASNI library, as it uses the same wrapper, only a recompilation with the `MODE_DMA` flag. In this approach, we use two continuous memory buffers, one to store the ASNI tailored descriptors and the other to store the packets in a contiguous fashion. The two buffers are allocated in host memory and are therefore accessible by the host like traditional memory. The ARM cores rely on the DMA API for reading and writing to these memory regions. The descriptor buffer acts as a ring of ASNI descriptors. In addition, the descriptors include a flag indicating whether the corresponding entry is valid, and when valid an offset where to find the packet in the packet buffer. The ARM sets this flag to 1 to signal that the descriptor has been filled. The host therefore polls on the next entry until it becomes valid. After the host processes the entry, it resets the flag to 0. When the ARM cores receive packets, the ASNI DMA pipeline performs a DMA write to transfer the packet payloads to the host as a scatter-gather list. The ASNI DMA pipeline on the ARM cores then issues a DMA write to update the descriptors on the host, indicating the payloads are available. As the host cannot issue DMA request to the ARM memory, the ARM cores fetch the descriptors when wrapping around the buffer and verify their flags are cleared before overwriting them.

This first design offered poor performance (around 30 MPPS, while the reference design can achieve 110 MPPS). As stated in Section 4.4, the DMA engine on the ARM cores has been shown in previous work [57] to be less efficient than the NIC engines. We explored and prototyped an alternative solution in which the payloads bypass the ARM cores and are directly sent to the

host. The ARM cores would then only have to transmit the descriptors. Recent NVIDIA NICs can write incoming packets directly into registered buffers across various memory locations, ARM, host CPU, or GPU. Memory region objects represent these registered buffers in the hardware, which store address translation data and attributes for each memory area. Each memory region is associated with a memory key. The NIC can write packets directly to the specified location by supplying the correct memory key and address. This latter implementation provides a small performance benefit as only the descriptors need to transit through the DMA API, at the cost of breaking payload continuity in memory, therefore not achieving the ASNI design. We therefore discarded the DMA-based implementations. We did not implement a TX mechanism using DMA as the RX was already underperforming. Some other NICs might perform better with this design than the BlueField 3. For instance, Humphries et al. [18] showcase DMA feature on an Intel Mount Evans SmartNIC.

In addition to low performance, the DMA implementations exposed a bigger attack surface. In the Ethernet design, we transmit normal Ethernet frames. Therefore, ASNI frames pose no greater threat than receiving a standard Ethernet packet. Since no pointers are passed from the ARM to the host, we use the size of the packets in the descriptor to retrieve the individual packets from the ASNI frame. An ASNI descriptor could contain an out-of-bound offset, which can be easily checked for in the ASNI wrapper for the application. However, in the DMA design where the ARM and the host exchange pointers, the ARM could pass a corrupted pointer to the host. Furthermore, the ARM can write to any addresses contained in the memory region exported by the host.

Received December 2024; revised April 2025; accepted April 2025