Your Programmable NIC Should be a Programmable Switch

Brent Stephens*
University of Illinois at Chicago

Aditya Akella
University of Wisconsin-Madison

Michael M. Swift
University of Wisconsin-Madison

ABSTRACT

Today's NICs are becoming programmable ("smart"). To support new network protocols, services, and offloads, there are NICs today that have on-board FPGAs, embedded processors, programmable forwarding pipelines, and specialized engines to support features like RDMA. Unfortunately, existing programmable NICs have a number of key limitations. It is difficult to chain offloads, schedule competing accesses to shared resources, and support functions that require variable processing time and thus may not run at line-rate.

In this paper, we propose PANIC, a new architecture for programmable NICs that overcomes the limitations of existing NIC designs. We divide the NIC into three components: 1) self-contained offload engines, 2) a logical switch, and 3) a logical scheduler. This design overcomes the limitations of existing designs and is able to scale with increasing line-rates to a large number of offloads and long offload chains.

1 INTRODUCTION

Networks are beginning to support complex offloads and in-network computation to accelerate applications, reduce the load on general purpose CPUs, and provide new complex and stateful network functions [13, 16–19, 21, 24, 31]. *Programmable ("Smart") NICs* have emerged as a new technology that is a key enabler of these new offloads and network functions [13, 17, 18].

There are many different programmable NICs [2, 7, 8, 13, 17, 20, 22, 23, 28], and there is also a wide range of different NIC designs. For example, programmable NICs that use a pipelined design place a chain of offload engines (*e.g.*, an FPGA) as a separate part of the NIC that logically sits in between the NIC and its TOR switch [13, 22]. In this way, the programmable component acts as a bump in the wire. Another popular programmable NIC design is the manycore NIC architecture. Instead of forwarding all packets through every offload, in this architecture, packets are load balanced across many embedded CPU cores [7, 23, 28, 38]. Finally, some programmable NICs use an on-NIC reconfigurable match+action (RMT) pipeline that can be programmed [17]. This can be used to perform custom message parsing, steer flows to queues, and provide new efficient DMA interfaces.

Unfortunately, these NIC designs have key limitations:

- Pipeline designs have difficulty chaining offloads. Because the offload topology is linear, the topology of the offload chain must match the order in which every packet needs to use the offloads. Further, slow offloads can cause head-of-line (HOL) blocking.
- Manycore architectures require the use of an embedded CPU core to orchestrate packet processing, which incurs tens of microseconds of additional latency [13].
- The types of offloads that can be supported by programmable forwarding pipelines are limited because each pipeline stage must be able to finish processing a packet in a single cycle. For example, it is not possible to perform IPSec offloading with an RMT pipeline.

This paper presents the design of PANIC, a new NIC architecture that overcomes the key limitations of existing smart NIC designs. This design draws inspiration from recent work on designing reconfigurable (RMT) switches [3, 5, 9, 34, 35] while simultaneously addressing the issues that arise as a result of NICs being a different environment than switches.

PANIC divides up the NIC into three complementary components: 1) self-contained offload engines, and 2) a logical programmable RMT switch, and 3) a logical scheduler. Figure 1 provides a high-level illustration of this design. Engines generate packets, which are parsed and used to make routing decisions. Packets are routed and placed in per-engine scheduling queues and then read by on-NIC engines.

NICs need to support many different types of offloads. For example, despite their differences, embedded processors, network processors, FPGAs, custom ASICs, regular expression engines, and memory caches are all potentially useful offloads. In PANIC, all offloads are implemented as independent engines, *i.e.*, independent tiles in the on-chip network. This allows for a wide-range of offloads to be easily supported, even if individual offloads do not run at line-rate.

Even parts of the NIC that would not normally be thought of as offloads are implemented as engines in PANIC. For example, PANIC uses a DMA engine and PCIe engine to interface with the main processor. These engines are attached to the logical switch in the same way as the offload engines.

*Work done while at University of Wisconsin-Madison

HotNets-XVII, November 15–16, 2018, Redmond, WA, USA

ACM ISBN 978-1-4503-6120-0/18/11...$15.00
https://doi.org/10.1145/3286062.3286068

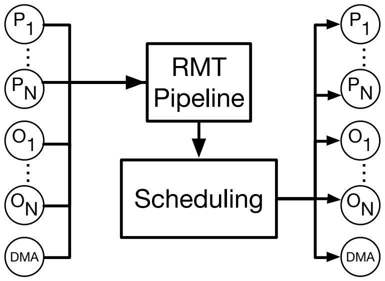


Figure 1: A logical overview of the PANIC architecture. Every Ethernet port (P_i), offload (O_i), and DMA/PCIe engine is connected to a common match+action (RMT) pipeline and scheduling pipeline. Each packet is processed and scheduled by a logically centralized pipeline as it is chained between engines even though the implementation of the logical switch and scheduler is distributed across the engines.

However, not every offload needs to process every packet. In PANIC, a logical programmable switch provides a common high-performance substrate for switching packets between the server, the network, and various offload engines. The logical switch is implemented through the coordination of a heavyweight RMT pipeline, lightweight lookup tables at each engine, and an on-chip network that interconnects the engines. This division of labor 1) avoids long wire lengths associated with building the logical switch with a single large crossbar and 2) avoids incurring the latency of the heavyweight RMT pipeline as packets are forwarded between offloads.

Lastly, each engine in PANIC also has a logical scheduler that determines the order in which the engine processes competing packets and requests. High-throughput applications using the same offloads as latency-sensitive applications can lead to performance isolation problems in NICs [40]. Packets in PANIC are inserted into these queues according to a slack time that is computed by the RMT pipeline and carried by the packets. This avoids these performance isolation problems.

This new architecture overcomes the limitations of existing NIC designs. Unlike pipelined designs, PANIC can dynamically chain packets between offloads without HOL blocking. Unlike manycore architectures, the logical switch is able to forward packets between different engines without requiring the involvement of CPU, which incurs high latency. Unlike RMT architectures, engines enable offloads that cannot be implemented directly as a part of an RMT pipeline.

2 PROGRAMMABLE NIC LIMITATIONS

We find that there are multitude of different offloads that can potentially improve application performance, although not every packet needs every single offload. However, all existing NIC designs suffer from key limitations with respect to supporting multiple independent offloads.

2.1 Types of Offloads

There is a vast potential for NIC offloads to accelerate application performance. Any computation performed to generate or process messages can be performed by the NIC.

Project	Offload Type
FlexNIC [17]	Application Inline Computation
Emu [37]	Application CPU-bypass Memory and Infrastructure CPU-bypass Network
SENIC [29]	Infrastructure Inline Network
sNICh [30]	Infrastructure CPU-bypass Network
DCQCN [41]	Infrastructure CPU-bypass Network
TCP Offload Engines [26]	Infrastructure CPU-bypass Network
Uno [18]	Infrastructure CPU-bypass Network
Azure SmartNIC [13]	Infrastructure CPU-bypass Network
RDMA	Application Inline/CPU-bypass Network/Memory

Table 1: The different offload types used by prior work.

For the sake of discussion, we categorize NIC offloads in the following dimensions: **Infrastructure vs. Application Offloads**, **CPU-bypass vs. Inline offloads**, and **Computation vs. Memory vs. Network offloads**. Using these dimensions, we find that most of the different possible types of offloads already exist and all different types are potentially useful. Additionally, different types of offloads may be provided by different types of hardware devices. ASICs, GPUs, embedded CPUs, FPGAs, and network processors are all potentially useful for implementing different offloads. To illustrate this taxonomy, Table 1 shows the different kinds of offloads that are provided by different prior works.

2.2 Vision

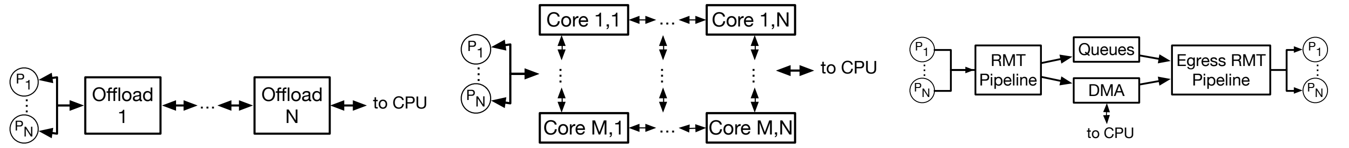
Any application that uses the network can potentially benefit from NIC offloads. However, different packets benefit from different sets of offloads, and every packet does not need every offload. This complicates on-NIC packet processing. An ideal programmable NIC should not restrict the type of offloads that may be simultaneously used. Instead, the NIC should be able to dynamically switch and schedule packets as needed between independent offloads.

For example, consider a key-value store like DynamoDB [36] that serves requests from multiple different tenants that may potentially be geodistributed across multiple data centers. Many different parts of this application can be offloaded, but not every packet needs every offload. For example, IPSec should be offloaded. However, only packets sent over the WAN need to be encrypted, so not every packet should be sent through the IPSec engine. Similarly, to reduce latency and CPU load, the NIC can cache the location of values for hot keys pairs and use DMA to directly return replies, completely bypassing the CPU. However, only requests that are cached on the NIC should be processed in this way. Requests that cannot bypass the CPU should instead be steered to appropriate receive queues used by the driver/application.

2.3 Current NIC Limitations

This section gives background information on the design of programmable NICs, all of which have key limitations.

2.3.1 Pipeline Designs. Figure 2a illustrates the pipelined programmable NIC design. In this design, the offloads are



(a) A pipelined programmable NIC architecture **(b) A tiled manycore programmable NIC architecture** **(c) A programmable NIC architecture with a reconfigurable match+action pipeline (like FlexNIC [17])**

Figure 2: Illustrations of existing programmable NIC architectures.

arranged in a linear sequence, *i.e.*, a pipeline. Effectively, each offload looks as though it is an independent device attached in the middle of the wire connecting the NIC to a TOR switch.

Pipeline designs are popular because they allow for offloads to be easily designed and built. Most existing NICs with on-board FPGAs located as a "bump-in-the-wire" use this design [12, 13, 22], and other NICs use this design for fixed function offloads for TCP checksums and IPSec [1, 15].

Pipeline designs suffer from two key limitations: 1) Packets are forwarded through offloads that do not need to process the packet. This increases latency and wastes on-NIC bandwidth. Further, this can cause HOL blocking with offloads that do not run at line-rate, although this can be avoided with logic to bypass offloads. 2) Chaining offloads is difficult because these designs lead to a static offload topology; the offloads are arranged in a line. Although it is possible to allow packets *recirculate* through the pipeline as needed, this is also wasteful of on-NIC bandwidth. If enough packets are recirculated, the NIC may not be able to process packets at line-rate.

2.3.2 Manycore Designs. Figure 2b illustrates a manycore programmable NIC design [8, 18, 23, 38, 39]. These designs implement network offloads by parallelizing flow processing across a large number of embedded processors that are arranged into a multi-hop on-chip network, (*i.e.*, a tiled topology). Some manycore NICs additionally contain hardware engines for cryptography and compression [38].

The main limitation of manycore NICs is that they use an embedded CPU core to orchestrate the processing of a packet. This is because the on-chip network cannot cannot parse complex packet headers to determine the appropriate on-NIC addresses for the packet's destination. Instead, manycore designs use a CPU to generate requests to hardware offloads as needed. However, when this is not needed, this can significantly increase latency. For example, Firestone *et al.* [13] report that processing a packet in one of the cores on a manycore NIC adds a latency of 10 μs or more.

2.3.3 Reconfigurable Match+Action (P4) Designs. Figure 2c shows a programmable NIC design using a programmable match+action (RMT) pipeline. This model was recently proposed by FlexNIC [17]. In this model, incoming packets are first parsed by a programmable parser and then sent through a pipeline of M+A tables.

The main limitation of RMT NIC designs is that they are limited in the functions that they can support. For example, RMT NICs cannot support compression, encryption, or any offload that must wait on the completion of a DMA from main memory. This is because the actions that are possible at each stage of the pipeline are limited to relatively simple *atoms* to guarantee that the entire pipeline can process packets at line-rate [34]. Unfortunately, many interesting offloads are too complex to fit in this model.

3 PANIC DESIGN

The core idea behind the design of PANIC is that the NIC should be implemented as three logical components :1) many different types of offload engines, 2) a logical switch, and 3) a logical scheduler. The logical switch is used to dynamically chain packets between offloads without additional orchestration. The logical scheduler is used to explicitly schedule when competing packets are processed at each engine. As a result of the logical switch and scheduler, there are no additional constraints placed on individual offload engines.

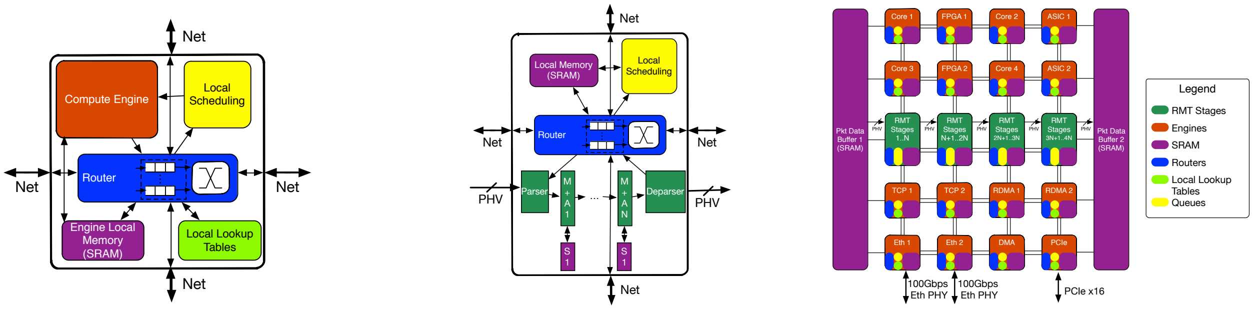
In this section, we describe the different components of PANIC and how they coordinate to implement a logical switch and logical scheduler in more detail. After that, we then give an example of the flow of packets in PANIC.

3.1 PANIC Components

A key insight of the PANIC design is that even messages between different on-NIC engines and offloads that are not Ethernet packets can be treated as if they were. For example, reading transmit descriptors, writing an incoming packet to main memory, and processing an RDMA request all need to use the DMA engine to read or write from main memory. In PANIC, these are all treated as packets. For the sake of clarity, we refer to both packets and engine-to-engine requests/response as *messages* from here on out.

This design allows PANIC to use a single unified on-chip network. If the messages sent between engines were treated differently than packets, then it would be necessary to implement two (or more) separate networks. While some manycore NICs have taken this approach and have as many as five separate on-chip networks [38, 39], using a unified network can lead to higher peak throughputs for a given aggregate network bit width[1].

[1] If there are multiple networks and one is in use while the other is not, then parallel wires are idle. If all of these wires were instead used for a single network, this could not be the case



(a) Offload Engine

(b) RMT Engine

(c) PANIC Overview

Figure 3: (a) An offload engine in the PANIC architecture. The compute engine and local memory are used by the offload. The router connects to neighbor engines to form an on-chip network. The local lookup tables allow for chaining messages between offloads without first passing through the RMT pipeline. The local request scheduling queues implement the logical scheduler. (b) An RMT engine that is used to build the heavyweight RMT pipeline. The difference between offload engines is that the compute engine and local lookup tables are replaced with a programmable parser and a match+action pipeline. (c) An illustration of the on-chip network that connects engines in PANIC. Together, these components cooperate to implement the logical switch and logical scheduler. The individual offload and RMT engines are connected in a tiled topology and form an on-chip message (packet) network.

3.1.1 Offload Engines. In PANIC, any component of the NIC that requires buffering or cannot run at line-rate is implemented as an engine attached to a common switch and scheduler (Figure 1). This includes the new components of programmable NICs like FPGAs, embedded processors, and ASICs for custom offloads. However, this also includes existing NIC components that would not normally be thought of as switch ports, including the on-NIC DMA and PCIe engines that are used to interact with main memory and the host CPU.

To connect to the on-chip network, each engine contains local lookup tables, a router, and scheduling queues. These additional components are used to implement the logical switch and scheduler and are illustrated by Figures 3a, 3b, and 3c.

3.1.2 Logical Switch. Due to physical constraints (*e.g.*, wire length), it is not feasible to build a single large switch and scheduler when there are a large number of engines [39]. To overcome this challenge, PANIC distributes the implementation of the logical switch and scheduler across the different engines, which are connected by an on-chip mesh network.

The *logical switch* in PANIC routes messages (packets) between the Ethernet ports, offloads, and the server. The switch has three components: a heavyweight RMT pipeline, lightweight lookup tables at every engine, and a unified on-chip network formed by connecting each engine to its neighbors in a mesh topology. Figure 3c illustrates this design, and we provide more details below.

Heavyweight RMT Pipeline: The heavyweight RMT pipeline parses complex message (packet) headers and performs stateful match+action processing. This includes load-balancing messages across descriptor queues, determining the chain of offloads that a message should be forwarded through, and computing slack times for the logical scheduler. Figure 3b illustrates the design of the RMT engines that are used to compose the heavyweight pipeline.

In PANIC, the heavyweight pipeline is composed of multiple RMT engines. Each engine contains a parser, multiple internal match+action stages, and a deparser. Neighboring engines may be configured to independently process messages or be chained to form a longer pipeline. This design allows for flexible trade-offs between pipeline depth and parallelism, with more pipelines leading to more throughput.

Most messages in PANIC must traverse the RMT pipeline at least once. Some messages may need to traverse the pipeline more than once, although the use of lightweight per-engine lookup tables is intended to minimize this. Ideally, PANIC is able to process unencrypted messages in one pass through the pipeline and encrypted messages in two passes.

Lightweight Lookup Tables: Lightweight lookup tables reduce the load on the heavyweight RMT pipeline and more evenly distribute traffic across the on-chip mesh network. When a message is processed by the RMT pipeline, instead of only looking up the next hop, a chain of engine destinations is found and added as a lightweight message header. These addresses are then matched on at each engine without requiring an additional heavyweight pipeline traversal.

Sometimes it is not possible to know the complete chain that is needed for a message. This is the case with encrypted messages. In this case, either a default route back to the heavyweight RMT pipeline is installed at the engine or the RMT pipeline includes itself as a nexthop in the chain so that it can generate the remainder of the chain.

Multi-hop on-chip networks: Instead of using a single crossbar to connect engines, PANIC uses a multi-hop on-chip network to forward messages between engines. Every engine contains a router, and the routers are connected in a 2D mesh topology. This is illustrated in Figure 3b. Every engine connects to its neighbors, including the RMT engines that compose the heavyweight RMT pipeline. The edges of the on-chip network are the engines that provide external interfaces, *e.g.*, the Ethernet ports and DMA/PCIe engines. Additionally, the on-chip network is lossless. If it is necessary to drop messages, this is done by the logical scheduler.

The on-chip network's performance can be characterized by its bandwidth and latency. The bisection bandwidth of the network scales with topology size due to multipathing. The routers add one cycle of latency at each hop, and the lightweight tables also add another cycle of latency.

3.1.3 Logical Scheduler. Every engine contains a local scheduling queue, and messages sent to each engine are placed in the queue local to the engine (Figure 3b). Together, these queues implement a logical scheduler that is used to avoid head-of-line blocking latency for high priority messages and ensure that messages from different applications, containers, and VMs share on-NIC resources according to some high-level policy.

In PANIC, each local scheduling queue is a priority queue. When the heavyweight RMT pipeline computes the chain of offloads to send a message through, it also computes an end-to-end slack time for each offload in the chain. This slack time is added as a header and determines the order in which messages are inserted into the priority queue for each engine.

Although simple, this approach is able to implement any arbitrary local scheduling algorithm [25]. As ongoing work, we are looking into how slack values should be computed so as to best enforce a high-level network policy.

3.2 Example

To understand the benefits of the PANIC design, consider the example offloads for a geodistributed multi-tentant KVS like DynamoDB in Section 2. When an Ethernet port is receiving incoming requests, they will be sent to the heavyweight RMT pipeline after being processed by the Ethernet MAC. The pipeline will parse the messages and use lookup tables to determine a chain of offloads to forward it to. If it is an IPSec packet, it will forward the packet to an IPSec engine, which will reinject the packet into the heavyweight RMT pipeline once it has finished decrypting the packet.

The heavyweight RMT pipeline will parse the request and determine a chain of engines to forward the message to. For GETs that should be sent to the main CPU, the pipeline will select a receive queue according to some policy and send the parsed request to a DMA engine. After the DMA has completed, the DMA engine will send a message to a PCIe engine that may generate an interrupt depending on the interrupt coalescing state. If the incoming request is a SET, the process is similar, except that the DMA engine will append the value in the SET to a log. However, if the request instead hits in the on-NIC application cache, it will be forwarded to an RDMA engine. This RDMA engine will then issue DMA requests (via the pipeline) to read the value, generate the packet headers for the response, and then inject this new response into the pipeline, where it will be switched to the Ethernet port for transmission after the heavy RMT pipeline deparses it.

Due to possible memory contention from applications on the main CPU, the DMA engine has variable performance and may become a bottleneck. However, the PANIC design is still able to avoid queuing latency for high-priority messages. Because of the scheduler, it is possible to ensure that all of the dependent accesses required to process a high priority message are able to bypass other pending DMA requests.

Line-rate	# Eth Ports	PPS
40Gbps	2	240Mpps
40Gbps	4	480Mpps
100Gbps	1	300Mpps
100Gbps	2	600Mpps

Table 2: The packet per second (PPS) throughput needed for line-rate forwarding of minimal sized packets in both RX and TX directions for different line-rates and NIC port counts.

4 DISCUSSION

This section discusses important aspects and implications of the PANIC design in more detail.

4.1 Programming Model

Individual offload engines, the logical switch, and the logical scheduler in PANIC are programmable. PANIC allows for individual engines to be programmed in the most appropriate domain specific language. The heavyweight RMT pipeline and lightweight lookup tables are programmed similarly to how current RMT switches are programmed (*e.g.*, using P4 [4] and SDN). The logical scheduler is programmed by associating each message with a slack time, and the responsibility for computing the appropriate slack time is placed on the heavyweight RMT pipeline.

4.2 Switch Throughput

The throughput of the logical switch must be high enough to send and receive messages at line-rate while also chaining messages between offloads as needed.

Two aspects of the logical switch in PANIC can be performance limitations: the throughput of the heavyweight RMT pipeline and the throughput of the on-chip network used to route messages between engines. For the RMT pipeline, we target a throughput that is able to guarantee that every TX and RX packet can be processed at least once, even when using minimal sized packets at line-rate. For the on-chip network, we target a throughput that is able to forward every packet at line-rate through some average length of offload chain before the packets are forwarded to their destinations.

Table 2 shows the number of minimally sized packets per second (pps) that can be transmitted and received given different NIC line-rates and port counts. For comparison, given a clock frequency of F and P parallel pipelines, the heavyweight RMT pipeline in PANIC can process $F * P$ packets per second. (Two 500MHz pipelines can process packets at a rate of 1000Mpps.) To be able to drive line-rate, the heavyweight RMT pipeline's throughput must be equal to or greater the NIC's line-rate (Table 2) multiplied by the average number of times each packet is processed by the pipeline.

Supporting today's increasing Ethernet line-rates in PANIC should not be problem. With two RMT pipelines and a

Line- rate	Freq	Bit Width	Topo	Bisec BW	Chain Len
40Gbps x2	500MHz	64	6x6 Mesh	384Gbps	5.60
40Gbps x2	500MHz	64	8x8 Mesh	512Gbps	8.80
100Gbps x2	500MHz	128	6x6 Mesh	768Gbps	3.68
100Gbps x2	500MHz	128	8x8 Mesh	1024Gbps	6.24

Table 3: The all-to-all network throughput and the number of offloads that a packet can be forwarded to in a chain assuming uniform traffic patterns for different on-NIC topologies.

500 MHz clock frequency, PANIC can forward every packet through the RMT pipeline at least once and still sustain line-rate even for a two port 100 Gbps NIC.

However, this analysis also demonstrates the need for the PANIC architecture. If the RMT pipeline is needed to switch packets between every offload, then it would not be possible to send each packet to even a single offload and sustain line-rate given a two port 100 Gbps NIC and two RMT pipelines running at 500 MHz.

Next, Table 3 shows that it should be possible to use a 2D mesh topology to route packets between offloads once the RMT pipeline has chosen an offload chain. In this analysis, we look at the topological properties of 2D mesh networks assuming that packets are uniformly distributed across offloads [10, 11]. For different topologies, frequencies, and channel bit widths, we compute the network's bisection bandwidth, capacity (all-to-all throughput), and average number of offloads that each packet may be forwarded to while still sustaining line-rate in both transmit and receive directions. Across a range of topology sizes, we find that reasonable clock frequencies and bit widths can support long average packet chain lengths.

4.3 Memory Pressure

Finally, PANIC does not increase memory pressure in the NIC, which is important as packet buffer space is a limited resource [29]. Although offloads that do not run at line-rate must buffer and eventually drop or pause traffic if packets using the offload arrive faster than the service-rate, this is fundamental to all programmable NICs. Nothing about PANIC increases memory pressure compared to the other programmable NIC designs in Section 2. Further, PANIC introduces mechanisms unavailable in other designs that can be used to intelligently drop packets when memory pressure is a limiting factor.

5 RELATED WORK

This section briefly discusses related work.

The architecture of the Tile-GX NICs [38] is most similar to PANIC. These NICs contain up to a hundred embedded CPU cores, a programmable packet parsing engine, and ASIC offloads for cryptography and compression. The principle limitation of the Tile-GX architecture is that it requires using an embedded CPU core to orchestrate packet processing.

PANIC builds upon recent work on the design of programmable switches [5, 6, 9, 27, 34]. Also, the logical scheduler in PANIC is similar to recent work on programmable packet scheduling [25, 35]. One of the contributions of PANIC is in adapting these designs to NICs, an environment with different requirements and constraints.

Additionally, PANIC is also an interesting point in the switch design space. Some reconfigurable switches provide functionality through an RMT pipeline [5, 34], while others use a collection of different engines [9]. PANIC introduces a new middle ground between these two competing designs.

Recent work has argued that the processing stages of a NIC should be modeled as a directed graph [14, 32, 33]. Similarly, prior work has proposed placing a switch on the NIC [13, 18, 30]. PANIC takes both of these ideas a step further by using an on-NIC switch to connect the different NIC components and offloads.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we argue that programmable NICs need to internally implement a programmable switch that connects the Ethernet ports, the offloads, and the PCIe connection to the server. We propose PANIC, a new NIC with an on-chip logical switch and logical scheduler that are able to route and schedule packets between different internal offload engines.

This approach overcomes the limitations of previous designs. PANIC supports arbitrary types of offloads. Messages are not unnecessarily sent to engines, and offloads that do not run at line-rate do not cause head-of-line blocking for packets that do not use the offload. PANIC is also able to chain packets between offloads without requiring either an embedded CPU core to orchestrate or requiring heavyweight RMT processing after each hop in the offload chain. We show that PANIC is able to scale performance with increasing line-rates, number of offload engines, and offload chain lengths given reasonable clock frequencies and bit widths.

There are still some open questions that are raised by the PANIC design: What is the best way to simultaneously provide lossless forwarding to ensure that important messages like DMA requests for descriptors are never dropped while also providing lossy forwarding to ensure that other messages (*e.g.*, packets from a DOS attack) are dropped as needed? What is the best way to provide flow control for lossless forwarding so that neither the heavyweight RMT pipeline nor the on-chip network are ever stalled by a slow or overloaded engine? What is the best on-chip topology? How should different engines be placed in this topology? Should entire packets always be passed from engines, or are there times when it is better to instead pass pointers to packet data located in a common packet buffer? We hope to answer these questions as part of our ongoing work.

Acknowledgments: We would like to thank the anonymous reviewers for their thoughtful feedback. Brent Stephens, Aditya Akella, and Michael Swift are supported in part by the NSF grant CNS-1717039.

REFERENCES

[1] Intel ethernet switch fm10000 datasheet. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/ethernet-multi-host-controller-fm10000-family-datasheet.pdf.

[2] Accolade Technology. Accolade ANIC. https://accoladetechnology.com/whitepapers/ANIC-Features-Overview.pdf.

[3] Barefoot. Barefoot Tofino. https://www.barefootnetworks.com/technology/#tofino, 2017.

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM CCR*, 2014.

[5] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. A. Mujica, and M. Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *SIGCOMM*. ACM, 2013.

[6] L. D. Carli, Y. Pan, A. Kumar, C. Estan, and K. Sankaralingam. PLUG: Flexible lookup modules for rapid deployment of new protocols in high-speed routers. In *SIGCOMM*. ACM, 2009.

[7] Cavium Corporation. Cavium CN63XX-NIC10E. http://cavium.com/Intelligent_Network_Adapters_CN63XX_NIC10E.html.

[8] Cavium Corporation. Cavium LiquidIO. http://www.cavium.com/pdfFiles/LiquidIO_Server_Adapters_PB_Rev1.2.pdf.

[9] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall. dRMT: Disaggregated programmable switching. In *SIGCOMM*. ACM, 2017.

[10] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2003.

[11] W. J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Trans. Comput.*, 39(6), June 1990.

[12] Exablaze. ExaNIC V5P. https://exablaze.com//exanic-v5p.

[13] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *NSDI*. USENIX Association, 2018.

[14] M. Flajslik and M. Rosenblum. Network interface design for low latency request-response protocols. In *USENIX ATC*, 2013.

[15] Intel. Intel 82599 10 GbE controller datasheet. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf.

[16] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing key-value stores with fast in-network caching. SOSP. ACM, 2017.

[17] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. High performance packet processing with FlexNIC. In *ASPLOS*. ACM, 2016.

[18] Y. Le, H. Chang, S. Mukherjee, L. Wang, A. Akella, M. M. Swift, and T. V. Lakshman. UNO: Uniflying host and smart NIC offload for flexible packet processing. In *SoCC*. ACM, 2017.

[19] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *OSDI*. USENIX, 2016.

[20] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *OSDI*. USENIX, 2016.

[21] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. Incbricks: Toward in-network computation with an in-network cache. In *ASPLOS*. ACM, 2017.

[22] Mellanox Technologies. Innova - 2 Flex Programmable Network Adapter. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf.

[23] Mellanox Technologies. Mellanox BlueField SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf.

[24] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *SIGCOMM*. ACM, 2017.

[25] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker. Universal packet scheduling. In *NSDI*. USENIX, 2016.

[26] J. C. Mogul. TCP offload is a dumb idea whose time has come. In *HotOS*. USENIX, 2003.

[27] R. Narayanan, S. Kotha, G. Lin, A. Khan, S. Rizvi, W. Javed, H. Khan, and S. A. Khayam. Macroflows and microflows: Enabling rapid network innovation through a split SDN data plane. In *EWSDN*. IEEE, 2012.

[28] Netronome. NFP-6xxx flow processor. https://netronome.com/product/nfp-6xxx/.

[29] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. SENIC: Scalable NIC for end-host rate limiting. In *NSDI*, 2014.

[30] K. K. Ram, J. Mudigonda, A. L. Cox, S. Rixner, P. Ranganathan, and J. R. Santos. sNICh: Efficient last hop networking in the data center. In *ANCS*. ACM/IEEE, 2010.

[31] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *NSDI*. USENIX, 2018.

[32] P. Shinde, A. Kaufmann, K. Kourtis, and T. Roscoe. Modeling NICs with Unicorn. In *PLOS*. ACM, 2013.

[33] P. Shinde, A. Kaufmann, T. Roscoe, and S. Kaestle. We need to talk about NICs. In *HotOS*. USENIX, 2013.

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

[35] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable packet scheduling at line rate. In *SIGCOMM*. ACM, 2016.

[36] S. Sivasubramanian. Amazon dynamoDB: A seamlessly scalable non-relational database service. In *SIGMOD*. ACM, 2012.

[37] N. Sultana, S. Galea, D. Greaves, M. Wojcik, J. Shipton, R. Clegg, L. Mai, P. Bressana, R. Soulé, R. Mortier, P. Costa, P. Pietzuch, J. Crowcroft, A. W. Moore, and N. Zilberman. Emu: Rapid prototyping of networking services. In *USENIX ATC*. USENIX, 2017.

[38] Tilera. Tile Processor Architecture Overview For the TILE-GX Series. http://www.mellanox.com/repository/solutions/tile-scm/docs/UG130-ArchOverview-TILE-Gx.pdf.

[39] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5), Sept. 2007.

[40] Y. Zhang, J. Gu, Y. Lee, M. Chowdhury, and K. G. Shin. Performance isolation anomalies in RDMA. In *KBNets*. ACM, 2017.

[41] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale RDMA deployments. In *SIGCOMM*. ACM, 2015.