StaRRNIC: Enabling Runtime Reconfigurable FPGA NICs

Anup Agarwal* Daehyeok Kim†
Srinivasan Seshan*

March 2023
CMU-CS-23-100

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Carnegie Mellon University, Pittsburgh, PA, USA
†University of Texas, Austin, TX, USA

Keywords: Programmable Networks, FPGA, NIC, Reconfigurable Hardware

Abstract

Programmability in the network has accelerated in-network applications like NATs, firewalls, caching, etc. However, much of this programmability is only compile-time. Outside changing a few configuration parameters or packet processing rules, changing the functionality requires taking the network element offline and reflashing it. We explore the use of FPGA partial reconfiguration primitives to reprogram network elements piece by piece without requiring to take the whole device offline. We identify key requirements such a solution must provide and find that existing work on enabling runtime reconfiguration does not fully meet these requirements. We explore potential ways to meet all the key requirements. We build and test a preliminary prototype on Alveo U280 FPGA board to validate the feasibility of using FPGA partial reconfiguration to provide runtime reconfiguration.

This report documents our preliminary study into the implementation of a runtime reconfigurable packet processing pipeline on an FPGA NIC. We hope this report provides useful guidance in designing similar artifacts or using our code (https://github.com/StaRR-NIC/starrnic-public, https://github.com/StaRR-NIC/xup_vitis_network_example/blob/starrnic/Notebooks/measure_exp.py).

1 Introduction

Recent advances in programmable ASICs [5, 8, 7], FPGAs [2, 4], and NPUs [1] have increasing the functionality that we can run on network switches and network interface cards (NICs) more sophisticated. The programmability enables them to run complex network functions–like NATs, firewalls, and load balancers [20, 25]–and to accelerate end-host applications [19, 27, 23] in addition to traditional functions like forwarding and filtering packets. With this trend, cloud service providers are deploying these advanced network devices in their production networks [3, 16, 6].

While NICs and switches have become programmable, until recently, they only provided *compile-time* programmability for the data plane. Before deploying a NIC or switch to serve traffic, a developer writes a program for the desired data plane functions. This is then compiled into a binary image (e.g., firmware for NICs) and flashed to the data plane. However, once the device is activated, it effectively becomes *fixed* in its function. Changing device programs requires careful planning ahead to avoid disruption, such as draining user traffic, provisioning capacity elsewhere, reflashing the devices, and finally reconnecting them.

Ideally, we want to be able to reprogram the data plane at runtime without requiring to take down the NIC/switch (for reflashing). While doing so, we do not want any significant packet loss or performance degradation during runtime reconfiguration (*zero downtime runtime reconfiguration*). Apart from this, for practical deployment, we also want reconfigurable NICs and switches to satisfy the following key requirements: (1) The performance and resource usage should be governed by the functions built by the developers and not by the control/management logic. (2) When performing runtime reconfiguration, the data plane logic must be updated consistently. We will discuss consistency models used in the packet processing context in §3.1. (3) For any stateful logic, the state must be preserved during and after reconfiguration. (4) To support concurrent applications from different tenants, they must support key properties required for multi-tenancy, including resource sharing and isolation during the reconfiguration period and normal operations. (5) As a single instance of the packet processing logic may not provide high performance, we want the ability to replicate logic to gain performance from parallelism.

While there are some recent attempts to address the limitation of compile-time programmability [34, 15, 31, 21], we observe that none of the existing work satisfies the all of the above requirements (§3.2).

We explore the design space of the runtime reconfigurable NIC architecture to understand what it takes to meet the above requirements. In particular, we focus on FPGA-based NICs that support partial reconfiguration (PR) that can be used for re-configuring a part of the NIC data plane.

We implemented a preliminary prototype on the Xilinx Alveo U280 FPGA board to verify the feasibility of using FPGA partial reconfiguration to support zero downtime runtime reconfiguration (§4). Our microbenchmarks show that our design can maintain its peak throughput without any packet loss during a reconfiguration period with a small spike in latency.

This report documents our preliminary study into the implementation of a runtime reconfigurable packet processing pipeline on an FPGA NIC. We hope this report provides useful guidance in designing similar artifacts or using our code (https://github.com/StaRR-NIC/starrnic-public, https://github.com/StaRR-NIC/xup_vitis_network_example/blob/starrnic/Notebooks/measure_exp.py).

2 Background

FPGA. Field Programmable Gate Arrays (FPGAs) are a family of hardware devices that can be reconfigured ("field-programmable"). To support this, an FPGA is composed of an array of generic logic blocks/gates, memory elements, and wires. A configuration memory region holds what computations the blocks perform and how they are interconnected by the wires. A firmware image or bitstream describes the configuration and can be loaded onto the FPGA to reconfigure its operation. During this reconfiguration the FPGA is not operational.

Partial Reconfiguration. Partial reconfiguration (PR) is a primitive that allows users to reconfigure a subset of FPGA logic at runtime, i.e., while the FPGA is operational. Users pre-define a *PR region (PRR)* or *dynamic region*. Users can then compile and load a bitsream to reconfigure just the circuit/logic inside the dynamic region at runtime. The circuit outside the dynamic region continues to operate as usual. The region that cannot be reconfigured at runtime is typically referred to as the static region.

During the PR operation, any logic running inside the PRR will be corrupted and any memory inside the PRR is lost. To protect against corrupt logic, typically the dynamic region is decoupled from the static region during the PR operation. This simply means that during the PR operation, the logic in the static region ignores the values of the signals originating from the dynamic region (as they may be corrupt).

For a more detailed background, we refer the reader to [30].

3 Design space exploration

Overview. For a runtime reconfigurable FPGA NIC, the design running on the FPGA will have (1) network functions (NFs) that users will run, and (2) logic to manage interaction between the network functions, the host server (PCIe interface), and the wire (NIC interface). The NFs will typically sit in the PR regions and the management logic will sit in the static region.

In our discussion, we define an NF as the processing functionality inside one PR region. The static design does not have visibility inside the NF (it cannot control how packets move through modules inside the NF). If multiple logical NFs sit inside a single PR region, we simply refer to these as sub NFs. The management logic and the NFs may also have both control plane logic and data plane logic. The control plane logic may be used for instance to collect telemetry data, or change packet forwarding behavior, etc.

We identify what features a runtime reconfigurable FPGA NIC may want to provide, and then we discuss how an implementation may provide these features.

3.1 Requirements

1. **Performance (throughput and latency) and resource efficiency.** We want the performance and resource consumption to be largely determined by the user provided network functions instead of the static management logic.

2. **Zero downtime runtime reconfiguration.** During runtime reconfiguration, we do not want any lost packets, loss in throughput, rise in latency, or any significant increase in resource consumption (resource consumption might increase as some reconfiguration strategies require resource headroom to program the new functionality).

3. **Consistent updates.** The literature has considered different consistency models. We classify them into two categories (per-packet, and across-packets) and summarize them here:

   (a) Per-packet. Consistency is defined based on what NFs process an individual packet.

   - Atomicity. If only a single NF is reconfigured, then we just want a packet to be processed by the old NF or the new NF. A packet should not be processed by an intermediate garbled NF.
   - Packet consistency [28]. If multiple NFs need to be reconfigured at the same time, we want a packet to be processed by either the old NFs or the new NFs. It shouldn't be the case that a packet is processed by one old NF and one new NF. If a packet is processed by one new NF and one old NF, this situation provides atomicity but not packet consistency.
   - Suffix causal consistency [24]. This is a weaker consistency model than packet consistency. A packet may be processed by an old NF followed by a new NF (but not the other way round).

   (b) Across-packets. Consistency is defined based on what NFs process potentially multiple packets [34]. These become interesting when there are potentially multiple control paths through the NFs.

   - Program consistent. If a packet is processed by a new NF, then all subsequent packets must be processed by only the new NFs. Note, in packet consistency, a packet may be processed by old NFs even if some other packet has been processed by new NFs. This is stronger than packet consistency.
   - Element consistent. If a packet is processed by a new NF, then all subsequent packets that would be processed by the new NF in the new program, must be processed by the new NF. This is weaker than program consistency, but different from packet consistency, i.e., there are scenarios that are element consistent but not packet consistent and vice versa.
     For instance, say there is only one control path and NF1 and NF2 need to be changed. If only one of the NFs is changed, the scenario is element consistent but not packet consistent. Now say both NFs are changed, but old copies of the NFs are also present in the system. If some packets go through old NFs and some through new NFs this is packet consistent but not element consistent.
   - Execution consistent. If a packet $\mathbb{P}$ is processed by a new NF, then all subsequent packets that traverse the same control path as $\mathbb{P}$ must be processed by the new NF.

4. **State or memory.** NFs may maintain some state. This state may also be shared between NFs (e.g., in a pipeline of NFs, upstream NFs may want to communicate results to downstream

NFs, or NF replica instances might want to share state for synchronization). State maintained by an NF may need to be preserved across the reconfiguration.

5. **Multi-tenancy.** Runtime reconfiguration enables applications like temporal and spatial multiplexing of different network functions and dynamic resource allocation to different network functions. This multiplexing is a useful feature for multi-tenant cloud environments that might share FPGA hardware for network functions (and potentially other non-networking applications). We list additional features that the FPGA design may want to provide for such a setting. These are motivated from [31, 22].

   (a) Resource sharing/isolation. NFs from different tenants might be deployed on the same FPGA NIC. For efficiency we want NFs to share resources (computational or memory) whenever possible. We also want them to be isolated (performance/security). For instance, a slow NF should not slow down packet processing of another tenant. An NF should not to be able to read/write the state of an NF from another tenant.

   (b) Sharing/isolation during runtime reconfiguration. The resource sharing and isolation properties described above should not be violated before, during, or after a reconfiguration operation.

6. **Replication or support for non-line-rate NFs.** Not all network functions may support high throughput. To support line rate performance, such network functions may be replicated [21]. Depending on the tradeoff choice, to simultaneously support zero downtime reconfiguration, consistent updates, and isolation/sharing, the static design may need to be explicitly aware of replicas. We discuss this further in §3.2 ("lose performance" in bullet 2).

3.2 Meeting the requirements

We discuss how the requirements might be met and how existing work meets them. We summarize this in Table 1.

1. **Performance (throughput and latency) and resource efficiency.** A number of prior works use *overlays* to provide partial reconfiguration. While FPGAs provide the abstraction of logical blocks, gates, memory, and wires; overlays implement a higher level abstraction (think of them as processors that can be configured or run instructions provided by users). Users then reprogram (or reconfigure) the overlay (or the processor) by providing new instructions (or configuration). FPGA PR changes the configuration of the logic blocks, gates etc.; overlay PR changes the configuration (instructions) of the overlay processor. Overlays are used by [34, 15, 31] to provide runtime reconfiguration.

   Typically, generating a bitstream for a PRR is time-consuming. The compiler needs to place and route logical hardware description onto the physical blocks on the FPGA. Generating configuration of the overlay however is quick. Typically, is no placement/routing involved. The configuration of the overlay is like a software program that describes what sequence of instructions need to be performed.

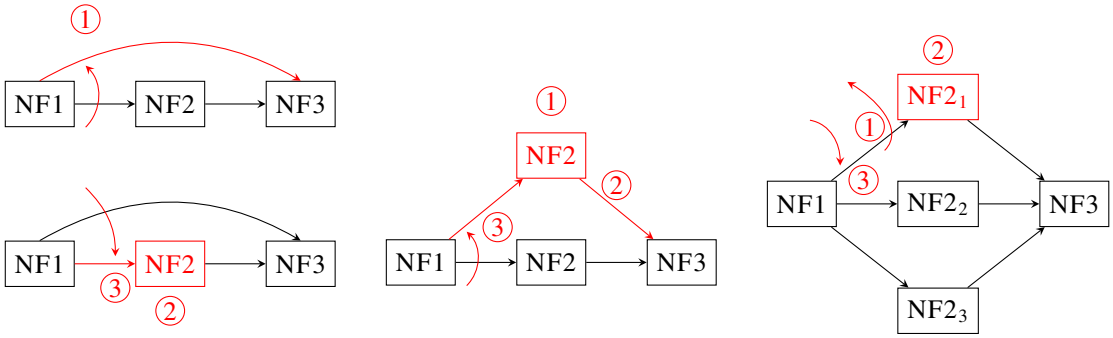
	Menshen [31] (based on RMT [10])	IPSA [15]	PANIC [22]	Rosebud [21]	FlexCore [34] (based on dRMT [11])
Performance or Resource efficiency	Limited by overlay	Limited by overlay	✓	High resource overhead (all NFs need to be replicated)	Limited by overlay
Zero downtime runtime reconfiguration	✓ (Resource headroom)	✓ (Resource headroom)	✗	✓ (Lose performance)	✓ (Resource headroom)
Consistent updates	✗	✗	✗	✓ (all NFs are changed together)	✓
Stateful NFs	✓	✓	✓	✓	✓
State sharing between NFs	✗	✓ (high latency)	✗	✓	✓ (high latency)
Isolation	✓ (line-rate by default due to overlay)	✓ (line-rate by default due to overlay)	✓	✗ (need to manually schedule between NFs (accelerators))	✗ (run-to-completion, no packet reordering)
Resource sharing	✓	✗	✗	✗	✗
Replication or support for non-line-rate NFs	✗	✗	✓	✓	✓

Table 1: Summary of requirements and prior work.

However, overlays come at a cost. Functionality is limited by the abstractions provided by the overlay. As a result an NF implemented using the overlay's abstraction might consume more resources and have poorer performance compared to one implemented directly using the FPGA's programming abstraction.

We can get the best of both worlds (fast compilation, and resource efficiency and performance) by having separate optimized overlays for different types of NFs, e.g., we can have one overlay that implements telemetry, and another overlay that implements intrusion detection, etc.. Users can then quickly reconfigure the overlays if they want to make small changes to the running NFs (using overlay reconfiguration or overlay PR). For instance if an overlay implements a count-min sketch [13] for telemetry, the overlay may support reconfiguring the number of rows/columns of the sketch, the flow key used by the sketch, etc.. If the users want to completely replace the NF (e.g., replace a telemetry NF with an intrusion detection NF), they can do this using FPGA PR. The optimized overlays can be precompiled offline as a result the long FPGA compilation times are not part for the critical path of runtime reconfiguration. Note, concrete NFs can also be precompiled, but optimized overlays offer more flexibility.

2. **Zero downtime runtime reconfiguration.** In a PR operation, the area being reconfigured cannot be used to process packets. Due to this we face a choice between three things (also



(a) Lose functionality **(b)** Resource headroom **(c)** Lose performance

Figure 1: Tradeoff space to provide zero downtime runtime reconfiguration. NF2 is being reconfigured.

illustrated in Figure 1).

- Lose functionality. We can choose to lose the functionality provided by the area being reconfigured. To avoid any loss or degradation in performance (throughput, latency), we would need to bypass the area being reconfigured before the PR operation.
- Resource headroom. If we have some extra area on the FPGA that is unused (say scratch area), we can configure the new functionality in the scratch area. And then convert the old functionality into a scratch area. To support this, we would need to divert traffic from the old functionality area to the new functionality area.
- Lose performance. This is applicable when there are multiple instances of an NF running. We can divert packets to a subset of the replicas, we can then reconfigure the area of the remainder replicas. This is the choice made by [21].

  Note, to implement this choice, the static design needs to be aware of the replicas to be able to control traffic between them. If the replicas are inside an NF as sub NF instances (i.e., the replicas are all inside a single PR region), then the static design cannot control traffic between them.

All the above choices require us to be able to divert traffic away or towards specific NFs (or areas) on the FPGA. This is typically fulfilled by a configurable interconnect [15].

Interconnect. On chip interconnects have been investigated in the hardware community [14]. Different interconnect choices trade off resources, throughput, latency, flexibility (traffic patterns that can be supported).

From the point of view of PR, and supporting zero downtime runtime reconfiguration, we can afford the interconnect to have high latency. The latency differences between interconnect choices would be on the order of a few clock cycles (at most 10s of nanoseconds) which is insignificant compared to microseconds of network propagation delays. However, a property we'd like to have is flexibility, i.e., the traffic patterns the interconnect can support. As the

FPGA is reconfigured, the placement of NFs may evolve over time. To be able to redirect traffic between NFs for PR operations, the interconnect might need to support large volume of traffic between an arbitrary pair of NFs. The complexity of the interconnect grows with the number of input/output ports of the interconnect. In an FPGA NIC design the interconnect will have one I/O port for each NF (PR region), one I/O port for each NIC interface (wire), and one I/O port for each host interface.

Interconnects that can support arbitrary permutation of traffic between their ports are called *non-blocking* interconnects (e.g., clos [12], crossbar, benes [9], batcher-banyan [26]). Blocking interconnects (e.g., ring, bus, tree, torus) on the other hand may not allow all permutations of traffic. If we consider different non-blocking interconnects, they tradeoff resources, latency, and potentially throughput (clock speed). A large clos topology will have lower resources, and higher throughput (clock speed) but higher latency than a crossbar for the same number of input output ports. However, in practice, NxN non-blocking interconnects (N inputs and N outputs), are implemented using multiple instances of 4x4 or 5x5 interconnects (typically crossbars). For small enough N, we do not get much resource benefits from the interconnects like clos compared to a crossbar. For today's FPGAs, we are able to instantiate 16x16 crossbars without much degradation in the clock speed (i.e., can support min sized packets at 100 Gbps linerate). This is large enough to support 12 PR regions, 2 NIC interfaces and 2 host interfaces. Given the area of today's FPGAs and the area it takes to implement meaningful network functions, we do not expect more than 10s of PR regions. Hence, crossbars suffice. It is an open question as to what is the best sweet spot between resources, latency, throughput, and flexibility is if/when the number of PR regions we can/wish to support increases in the future. A potential sweet spot is proposed by [15]. They reduce flexibility by clustering NFs and then only allow rerouting between NFs within the same cluster.

A configurable interconnect design not considered by prior work (to the best of our knowledge) is to have the interconnect sit inside a PR region. The interconnect then just hard codes the connections between its inputs and outputs. When the on-chip routing needs to be changed, the hard coded wires can be changed using a PR operation. Note, we cannot take the interconnect offline, so to support a PR operation on the interconnect, we would have two instances of the interconnect. To swap out the interconnect, we would need to divert the traffic to the other interconnect before performing the PR operation on the interconnect. Then we can redirect traffic to the new interconnect. To support redirecting traffic between the two interconnect interfaces we can use a simple crossbar interconnect (this is fine as we only need few ports on the crossbar).

In the hardware literature, another important aspect of interconnects is the routing protocol used by the interconnect. We can simplify this in the networking context, by having routes computed offline on the host server as opposed on online on the FPGA. Typically, the set of NFs that need to process different flows is known. Thus, we can segregate flows by the set of NFs that need to process the flows. Then we can compute routes for each class of flows offline and install routing rules in the interconnect. These rules can be changed as traffic/requirements evolve.

3. **Consistent updates.** We want to provide ensure updates are consistent when multiple NFs (PRRs) need to be reconfigured simultaneously. If we can only reconfigure one PRR at a time, then multiple NFs need to be reconfigured one-by-one. As a result there may be periods when one NF has been reconfigured, but another NF has not. Such cases can violate packet consistency. To ensure consistency isn't violated, we need to keep around the old NF configurations until all NFs (that need to be reconfigured simultaneously) have been reconfigured. To keep around old NFs, we need some resource headroom, and we also need a way to route packets through old and new NFs (this is done through versioned routing).

   Versioning. This is described in [28] (*two-phase update*). Specifically, packets are tagged with a version number. Packets are routed based on their version number. New NFs can be installed with newer version numbers (these are *unobservable* by the packets without the new version number). When all the new NFs are configured, packets can then be tagged with the new version number (this is a one-touch update in the terminology of [28]).

4. **State or memory.** Any memory inside the PR region is lost. If we want state to be preserved, this memory needs to sit in the static design (outside the PR regions).

   One option is to have the memory right next to the PR region and only allow memory accesses from the function inside the associated PR region. With this design the preserved state cannot be directly shared between the NFs, and if the NF is relocated due to PR (this happens in the resource headroom approach to achieve zero downtime runtime reconfiguration), then it cannot access the state.

   Another option is to have an interconnect between NFs and memory modules. This way any NF can access any memory region. The downside of this approach is that memory access time is increased. This can potentially stall the throughput of read after write operations. Possible solutions include: (1) have a cache that sits close to the NFs (can also be inside the PR region), (2) use processing-in-memory, i.e., have some generic compute logic collocated with the memory (e.g., increment by constant, add data in two addresses), the NFs then send memory manipulation instructions over the interconnect. Such solutions are present in the Netronome Agilio NICs [1].

5. **Multi-tenancy.**

   (a) Resource sharing. Menshen [31], through its use of overlays, allows sharing compute resources between NFs. Depending on the packet headers (flow ID), the overlay can run different instructions. If NFs are directly programmed on the FPGA (without an overlay), they may be performing arbitrary computation and there is no way to share the area with some other NF.

   In our vision of overlays that are optimized for different overlays (§3.2 bullet 1), resources can be shared between NFs of the same type (overlay). For instance, the telemetry NFs from two tenants can share resources. The flow key and measurement statistic could be specified by the overlay configuration, and computation/measurement is done by the same physical resources for the two NFs.

If an overlay is shared and a tenant wants to replace the overlay used by the NF (e.g., move from a telemetry NF to a firewall NF), then the sharing cannot continue. The new NF would need to be placed in a new PR region or it can be shared with another NF that uses the firewall overlay (from potentially a third tenant).

(b) Isolation. There may be cases where some NFs are slow. If the packets are processed in FIFO order, then these slow NFs might slow down the throughput of packets that never need to be processed by this slow NF (head-of-line (HoL) blocking). [22] provides isolation in these settings by scheduling packets to NFs. If the NF that should process a packet is busy, the packets go into a scheduling buffer. This design may limit performance. The scheduler may need to schedule packets to multiple NFs in the same cycle and the interconnect might limit how many packets can be sent from the scheduler in one cycle. An alternate design is to have a scheduler (priority queue, e.g., PIFO [29]) before every NF.

There may be cases when an NF processes packets quickly for one tenant (say tenant $\mathbb{A}$) but slowly for another tenant (say tenant $\mathbb{B}$). In this case, packets of tenant $\mathbb{B}$ may fill the priority queue. When this happens, the priority queue should drop packets in priority order to avoid HoL blocking, i.e., drop tenant $\mathbb{B}$'s packets whenever tenant $\mathbb{A}$'s packets arrive and the queue has $\mathbb{B}$'s packets [18].

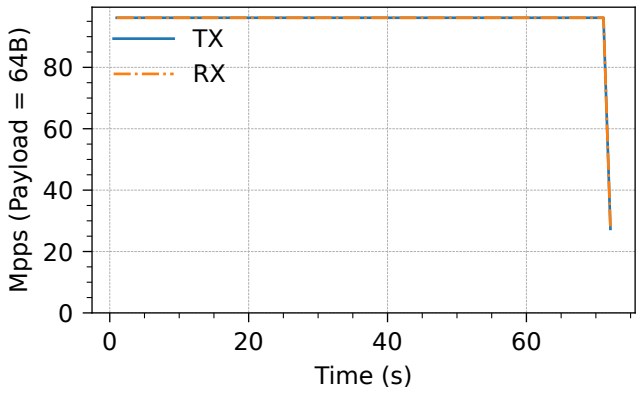
4 Prototype Implementation & Microbenchmarks

We perform microbenchmarks to verify if there are any unanticipated challenges associated with partial reconfiguration and meeting the zero downtime reconfiguration requirement. We implement a basic prototype that allows bypassing a network function and reconfiguring it (loss of functionality tradeoff from §3.2). We measure throughput and latency during reconfiguration.

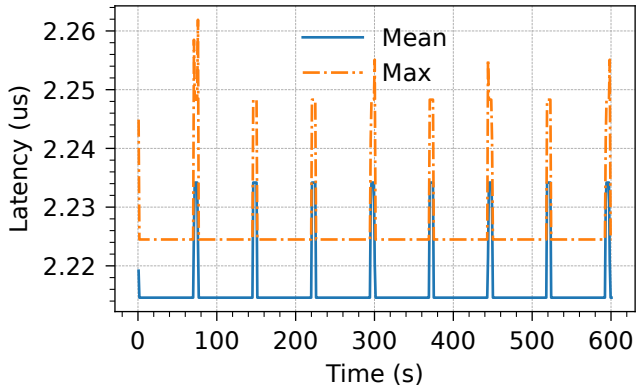
Setup. Our testbed consists of two FPGAs (Xilinx Alveo U280 boards [2]). One FPGA serves as a traffic source and sink. The other FPGA is the device-under-test (DUT) on which we perform the PR operation. We use [33] to implement the traffic source/sink. Since our source/sink are on the same device, we can accurately measure round trip latency. Our DUT code builds upon/uses the projects: Open NIC [32], and Corundum [17].

We compile a static design with a single PR region, we compile two NFs that can fit into the PR region and use PR operations to change the NF running. The operation involves diverting packets away from the PR region (this is done using a demultiplexer), then reconfiguring the PR region, and finally diverting traffic back to the PR region. We also ensure that during reconfiguration, the static design does not look at the signals coming from the PR region, we do this using a multiplexer attached to each signal originating from the PR region.

We use UDP packets with 64B payload. This is the minimum packet size that our traffic source/sink supports. This translates to 64 + UDP (8), IP (20), Ethernet(14) and FCS (4) = 110B frames, or 110 + IFG (12), preamble (7), start frame delimiter (1) = 130B on the wire. On a 100Gbps link, the theoretical maximum packet rate for this packet size is 100 * 1000 Mbps / (130 * 8 bits per packet) = 96.15 Mpps.



(a) Throughput. The experiment is ended at 70 seconds.



(b) Latency

Figure 2: Performance during PR operations.

We perform PR over the JTAG interface on the FPGA board. The operation involves opening Vivado hardware manager, reconfiguring the multiplexers and demultiplexers, and finally loading the partial bitstream that reconfigures the PR region. This process takes roughly 32.5 seconds. When a PR operation finishes, we invoke another one after 5 seconds. So we get a PR operation every 37.5 seconds. [21] implements PR over PCIe, which is faster and does not require launching the Vivado hardware manager.

To measure throughput, we continuously send packets from the source and then at the sink keep track of how many packets arrived by any given time. The throughput is calculated as (total packets arrived/time since start of experiment). To measure latency, we send a probe packet every 50 clock cycles (the benchmarking design has 294 Mhz clock). The benchmarking tool maintains a counter incremented every clock cycle. The counter value is injected into the packet payload and compared with the counter value at packet receive time to obtain the packet round trip time (latency).

Observations. We show throughput and latency during PR operations in Figure 2. We observe that throughput is not affected by PR operations. We observe a small latency spike every other PR operation. We do not exactly know the cause of this. We suspect this is due to delay added in reconfiguring the multiplexers/demultiplexers. There is no packet loss.

Acknowledgments

We thank Chris Neely, Gordon Brebner, and Xilinx, Inc. for providing us with the FPGA hardware, early access to their P4 to FPGA toolchain, and technical support. We thank Zhipeng Zhao, Shashank Obla, James Hoe, Alex Forencich, Moein Khazraee, Hugo Sadok, and Nirav Atre for useful discussions.

Availability

The code for this project is available at https://github.com/StaRR-NIC/starrnic-public. Our benchmarking code is available at https://github.com/StaRR-NIC/xup_vitis_network_example/blob/starrnic/Notebooks/measure_exp.py.

References

[1] December 2020. [Online; accessed 3. Apr. 2023]. URL: https://www.netronome.com/media/redactor_files/WP_NFP_Programming_Model.pdf.

[2] Alveo U280 Data Center Accelerator Card, April 2023. [Online; accessed 3. Apr. 2023]. URL: https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.

[3] AWS Nitro System, March 2023. [Online; accessed 3. Apr. 2023]. URL: https://aws.amazon.com/ec2/nitro.

[4] Intel® FPGA PAC N3000, April 2023. [Online; accessed 3. Apr. 2023]. URL: https://www.intel.com/content/www/us/en/products/sku/193920/intel-fpga-pac-n3000/specifications.html.

[5] Intel® Tofino™ Series Programmable Ethernet Switch ASIC, April 2023. [Online; accessed 3. Apr. 2023]. URL: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[6] Introducing C3 machines with Google's custom Intel IPU | Google Cloud Blog, April 2023. [Online; accessed 3. Apr. 2023]. URL: https://cloud.google.com/blog/products/compute/introducing-c3-machines-with-googles-custom-intel-ipu.

[7] Spectrum-4 Datasheet, April 2023. [Online; accessed 3. Apr. 2023]. URL: https://resources.nvidia.com/en-us-accelerated-networking-resource-library/ethernet-switches-pr?lx=LbHvpR&topic=Networking%20-%20Cloud#page=1.

[8] Trident4 / BCM56880 Series, March 2023. [Online; accessed 3. Apr. 2023]. URL: https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series.

[9] V. E. Beneš. On rearrangeable three-stage connecting networks. *The Bell System Technical Journal*, 41(5):1481–1492, 1962. doi:10.1002/j.1538-7305.1962.tb03990.x.

[10] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 99–110, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2486001.2486011.

[11] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. Drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 1–14, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3098822.3098823.

[12] Charles Clos. A study of non-blocking switching networks. *The Bell System Technical Journal*, 32(2):406–424, 1953. doi:10.1002/j.1538-7305.1953.tb01433.x.

[13] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005. URL: https://www.sciencedirect.com/science/article/pii/S0196677403001913, doi:https://doi.org/10.1016/j.jalgor.2003.12.001.

[14] William James Dally and Brian Patrick Towles. *Principles and practices of interconnection networks*. Elsevier, 2004.

[15] Yong Feng, Zhikang Chen, Haoyu Song, Wenquan Xu, Jiahao Li, Zijian Zhang, Tong Yun, Ying Wan, and Bin Liu. Enabling in-situ programmability in network data plane: From architecture to language. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 635–649, Renton, WA, April 2022. USENIX Association. URL: https://www.usenix.org/conference/nsdi22/presentation/feng.

[16] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, April 2018. USENIX Association. URL: https://www.usenix.org/conference/nsdi18/presentation/firestone.

[17] Alex Forencich, Alex C Snoeren, George Porter, and George Papen. Corundum: An open-source 100-gbps nic. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 38–46. IEEE, 2020.

[18] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2342356.2342358.

[19] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132764.

[20] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, SOSR '16, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2890955.2890968.

[21] Moein Khazraee, Alex Forencich, George C. Papen, Alex C. Snoeren, and Aaron Schulman. Rosebud: Making fpga-accelerated middlebox development more pleasant. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 586–605, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3582016.3582067.

[22] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 243–259. USENIX Association, November 2020. URL: https://www.usenix.org/conference/osdi20/presentation/lin.

[23] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3341302.3342079.

[24] Sheng Liu, Theophilus A. Benson, and Michael K. Reiter. Efficient and safe network updates with suffix causal consistency. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3302424.3303965.

[25] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 15–28, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3098822.3098824.

[26] M.J. Narasimha. The batcher-banyan self-routing network: universality and simplification. *IEEE Transactions on Communications*, 36(10):1175–1178, 1988. doi:10.1109/26.7538.

[27] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A programming system for nic-accelerated network applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 663–679, USA, 2018. USENIX Association.

[28] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, page 323–334, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2342356.2342427.

[29] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 44–57, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2934872.2934899.

[30] Kizheppatt Vipin and Suhaib A. Fahmy. Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Comput. Surv.*, 51(4), jul 2018. doi:10.1145/3193827.

[31] Tao Wang, Xiangrui Yang, Gianni Antichi, Anirudh Sivaraman, and Aurojit Panda. Isolation mechanisms for High-Speed Packet-Processing pipelines. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1289–1305, Renton, WA, April 2022. USENIX Association. URL: https://www.usenix.org/conference/nsdi22/presentation/wang-tao.

[32] Xilinx. open-nic. [Online; accessed 29. Mar. 2023]. URL: https://github.com/Xilinx/open-nic.

[33] Xilinx. xup_vitis_network_example, April 2023. [Online; accessed 2. Apr. 2023]. URL: https://github.com/Xilinx/xup_vitis_network_example.

[34] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. Runtime programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 651–665, Renton, WA, April 2022. USENIX Association. URL: https://www.usenix.org/conference/nsdi22/presentation/xing.