# DHL: Enabling Flexible Software Network Functions with FPGA Acceleration

Xiaoyao Li[1] Xiuxiu Wang[1] Fangming Liu*[1] Hong Xu[2]
[1]Key Laboratory of Services Computing Technology and System, Ministry of Education,
School of Computer Science and Technology, Huazhong University of Science and Technology, China
[2]NetX Lab, City University of Hong Kong

***Abstract*—Network function virtualization (NFV) aims to run software network functions (NFs) in commodity servers. As CPU is general-purpose hardware, one has to use many CPU cores to handle complex packet processing at line rate. Owing to its performance and programmability, FPGA has emerged as a promising platform for NFV. However, the programmable logic blocks on an FPGA board are limited and expensive. Implementing the entire NFs on FPGA is thus resource-demanding. Further, FPGA needs to be reprogrammed when the NF logic changes which can take hours to synthesize the code. It is thus inflexible to use FPGA to implement the entire NFV service chain.**

**We present dynamic hardware library (DHL), a novel CPU-FPGA co-design framework for NFV with both high performance and flexibility. DHL employs FPGA as accelerators only for complex packet processing. It abstracts accelerator modules in FPGA as a hardware function library, and provides a set of transparent APIs for developers. DHL supports running multiple concurrent software NFs with distinct accelerator functions on the same FPGA and provides data isolation among them. We implement a prototype of DHL with Intel DPDK. Experimental results demonstrate that DHL greatly reduces the programming efforts to access FPGA, brings significantly higher throughput and lower latency over CPU-only implementation, and minimizes the CPU resources.**

## I. Introduction

Network function virtualization (NFV) aims to shift packet processing from dedicated hardware middleboxes to commodity servers [1]–[4]. Implementing network functions (NFs) in software entails many benefits, including lower equipment cost, better flexibility and scalability, reduced deployment time, etc [5], [6]. However, compared to dedicated network appliances, software NFs on commodity servers suffer from severe performance degradation due to overheads of the kernel TCP/IP stack [6]–[10].

To overcome this limitation, several high performance packet I/O engines are proposed. Intel DPDK [11] and netmap [7]) bypass the kernel network stack to obtain packets directly from the NIC to improve the I/O of software NFs. For example, DPDK only needs two CPU cores to achieve 40 Gbps throughput for L3 forwarding [12]. Although these packet I/O engines provide the high I/O, it is still resource-demanding to deploy NFs that have computation-intensive processing logic, such as IPsec gateway and Network Intrusion Detection System (NIDS). For example, it takes as many as 20 CPU cores to reach 40 Gbps when performing packet encryption with IPsec [13].

To further improve the performance of software NFs, there is an emerging trend to implement NFs on FPGA in both academia [14]–[17] and industry [18]. As a form of programmable silicon, FPGA enjoys high programmability of software and high performance of hardware. Yet most of current FPGA solutions implement the entire NF, including the complete packet processing logic, on the FPGA board to the best of our knowledge. This approach suffers from several drawbacks.

First, the FPGA-only approach is resource-wasting. NFs generally contain complex protocol processing and control logic, which consumes many resources when realized in hardware circuit. Besides, there are many types of NFs, this incurs very high deployment cost with much resource demanding. Second, developing and programming FPGA is time-consuming. It usually takes hours to synthesize and implement HDL (Hardware Description Language) source code to final bitstream [14]. Since debugging and modifying the design of NF is common in practice, the FPGA-only approach presents a significant barrier to fast development.

We present dynamic hardware library (DHL)[1], a CPU-FPGA co-design packet processing framework that enables flexible FPGA acceleration for software NFs. In DHL, FPGA serves as a hardware accelerator, not as a complete network appliance. Specifically, we extract the computation-intensive part, such as encryption and decryption in IPsec, pattern matching in intrusion detection, etc., and offload them to FPGA as accelerator modules. We provide a DHL Runtime that hides the low-level specifics of FPGA, and we abstract the accelerator modules as hardware functions in FPGA that NF developers can call just like calling a software function. DHL decouples the software development of software NFs and the hardware development of FPGA accelerator modules that software developer and hardware developer can concentrate on what they are capable at. In addition, it provides data isolation

*The corresponding author is Fangming Liu (fmliu@hust.edu.cn). This work was supported in part by the National Key Research & Development (R&D) Plan under grant 2017YFB1001703, in part by NSFC under Grant 61722206 and 61761136014 and 392046569 (NSFC-DFG) and 61520106005, in part by the National 973 Basic Research Program under Grant 2014CB347800, in part by the Fundamental Research Funds for the Central Universities under Grant 2017KFKJXX009, and in part by the National Program for Support of Top-notch Young Professionals in National Program for Special Support of Eminent Professionals.

[1] DHL is opensourced on github: https://github.com/OpenCloudNeXt/DHL.

that supports multiple software NFs to use accelerator modules simultaneously without interfering each other.

This paper describes the DHL architecture and makes the following contributions:

1) We propose DHL, a CPU-FPGA co-design framework which not only enables sharing FPGA accelerators among multiple NFs to maximize utilization but also eases the developing effort by decoupling the software development and hardware development.
2) We abstract accelerator modules in FPGA as hardware functions and provide a set of programming APIs, which enable software applications to easily leverage FPGA accelerators with minimal code changes.
3) We design a high throughput DMA engine between CPU and FPGA with ultra-low latency for network application. We achieve this by adopting several optimization techniques of batching, user-space IO and polling.
4) We build a prototype of DHL and implement it on a server equipped with a Xilinx FPGA board. Our evaluation on the prototype demonstrates that DHL can achieve similar throughput as high as 40 Gbps compared with previous FPGA-only solutions, while incurring reasonable latency less than 10 $\mu$s.

The rest of the paper is organized as follows. In Section II, we introduce the motivation and challenges of our work. Section III describes the overview of DHL. The implementation details is presented in Section IV. Then we evaluate our DHL system in Section V and discuss the improvement and applicable scenarios in Section VI. Next, we survey the related work in Section VII. Finally, the paper is concluded in Section VIII.

## II. Motivation and Challenges

### A. FPGA is Not Perfect

There is much work about using FPGA for packet processing. Whether it is a specific NF (e.g., a router [19], redundancy elimination [17], load balancer [20], etc.) or a framework to develop high-performance NFs [14], [21], [22], they all demonstrate the fact that a hardware-based design performs better than a software design with general purpose CPU.

There are usually many types of NFs. When it comes to FPGA, there are some vital limitations for implementing NFs.

Firstly, the programmable logic units (look-up tables, registers, and block RAMs) of an FPGA are limited. It is resource-wasting to put everything into FPGA as every module with specified functionality consumes dedicated logic units. Indeed, we can use a more advanced FPGA which contains more resources, but the price of high-end FPGA is very high. For example as of July 2017 a Xilinx UltraScale chip costs up to $45,711[2], which is 20× higher than that of a CPU ($2,280[3] for Intel Xeon E5-2696v4, 22 cores).

Secondly, it is not friendly to change existing logics. Although FPGA is programmable and there is much prior work [14], [21], [22] aiming at improving the usability, it still suffers from long compilation time. When new requirements and bug occur, it usually takes hours to re-synthesize and re-implement the code.

### B. CPU is Not That Bad

The emerging high performance packet I/O engines such as Intel DPDK [11] and netmap [7] bring a new life to the software NFs. As reported by Intel, just two CPU cores can achieve 40 Gbps line rate when performing L3 layer forwarding [12]. Further, we survey a wide variety of common network functions, and find that there are two types of processing in data plane:

- ***Shallow Packet Processing:*** Executing operations based on the packet header, i.e., L2–L4 layer headers, such as NAT, L2/L3 forwarding/routing, firewall, etc.
- ***Deep Packet Processing:*** Executing operations on the whole packet data, such as IDS/IPS, IPsec gateway, flow compression, etc.

For shallow packet processing, as the packet headers follow the unified protocol specification, it usually does not need too much computation efforts. For example, it just needs to check and modify the specified header fields. For most common operations – table lookup, we find that searching an LPM (longest prefix match) table takes 60 CPU cycles on average (26 ns@2.4 Hz), as shown in Table I. It is fairly fast for CPU to perform shallow packet processing.

However, in terms of deep packet processing, it usually needs much more compute resources since data of higher layers has no regular patterns. Regular expression match from deep packet inspection (DPI) [23] and cryptographic algorithms [24] in IPsec gateways usually take more CPU cycles to complete the processing (again see Table I).

TABLE I
Performance of DPDK with one CPU core

| Network Function | Latency (cpu cycles) with one core | Throughput |
|---|---|---|
| L2fwd | 36 | 9.95 Gbps |
| L3fwd-lpm | 60 | 9.72 Gbps |
| IPsec-gateway | 796 | 1.47 Gbps |

[1] L3fwd-lpm uses longest prefix match (lpm) table to forward packets based on the 5-tuple; IPsec-gateway uses AES-CTR for cipher and SHA1-HMAC for authentication;
[2] Test with 64B packets, Intel 10G X520-DA2 NICs, DPDK 17.05 on CentOS 7.2, kernel 3.10.0, with an Intel Xeon E5-2650 V3 @ 2.30GHz CPU.

### C. CPU-FPGA Co-Design & Challenges

Considering the pros and cons of FPGA and CPU, an intuitive solution is to combine them together that keeping the control logic and shallow packet processing running on CPU and offloading the deep packet processing to FPGA. Compared with FPGA-only NF solution, FPGA-CPU co-design brings four key benefits.

- First, it fully utilizes FPGA resources. Only the computationally intensive processing are offloaded to FPGA as accelerator modules, which generally has a reasonable

[2] The price is from AVNET, https://www.avnet.com
[3] The price is from Amazon, https://www.amazon.com

resource requirement of LUTs and BRAM. Thus an FPGA board can host more modules.

- Second, software NFs can flexibly use these FPGA accelerator modules on-demand, without having to re-implement them from scratch. This also saves a lot of FPGA development time and effort.
- Third, the accelerator modules can be standardized to perform standard compression and encryption/decryption operations, which do not need to change often. For example we can have a suite of accelerator modules for various categories of AES algorithms. The rest of the NF logic is in software, and can be flexibly and frequently changed without modifying the FPGA accelerator implementation.
- Last, it decouples the software development of NF and the hardware development of accelerators. This means that hardware experts can make the best efforts to optimize the accelerator design without considering the complexity of the NF, and software NF developers do not need to be aware of the details of accelerators.

Meanwhile, the requirements of high performance and high scalability pose several challenges to a CPU-FPGA co-design packet processing framework:

- Considering the high throughput and low latency requirement of network functions, it calls for a high throughput data transfer solution which is applicable for small packets (64B to 1500B) with minimal transfer delay between host and FPGA.
- When it comes to the multiple NFs situation, previous work [14], [19], [20] is not applicable, as they only target for one type of NF at a time. It should be a flexible framework that supports multiple accelerator modules for different NFs in an FPGA. Even more, in consideration of the limited resource, it should support reconfigure the accelerator modules without interfering other running modules on the fly.
- When multiple NFs call the same/different accelerator module in the same FPGA, how to ensure the data isolation between them and how to return the post-processed data to the right NF.

## III. System Overview

Here we give an overview of the DHL framework as shown in Figure 1. We design DHL as a generic framework to enable software NFs to easily access FPGA accelerators. In DHL, only hardware function abstractions are exposed to software developers in the form of a set of programming APIs, and the accelerator sharing and management are transparently handled by our DHL Runtime.

### A. Packet I/O Engine

The packet I/O engine determines the raw performance of the framework. We choose to use Intel DPDK [11] because it provides a set of comprehensive and efficient primitives to ease the development of high performance network packet processing framework. For example, its lockless multi-producer multi-consumer ring library is particularly useful for buffering data from different software NFs where we need to aggregate and distribute packets with minimal overheads. Its NUMA-aware hugepage memory management library and multi-core management library make it easy to built multi-socket and multi-core framework.

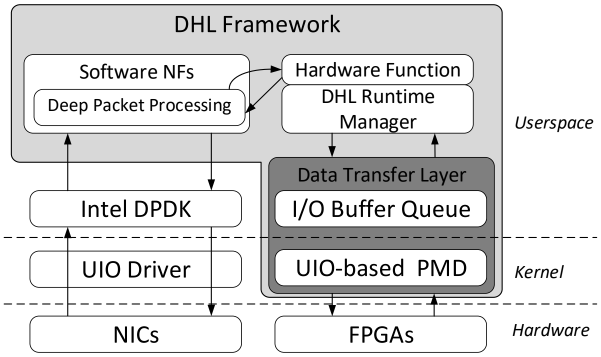


Fig. 1. Overview of DHL framework.

TABLE II
DHL programming APIs

| Functions | Description |
|---|---|
| DHL_register() | an NF registers itself to the DHL Runtime |
| DHL_search_by_name() | for the NF to query its desired hardware function |
| DHL_load_pr() | for the NF to load a partial reconfiguration bitstream |
| DHL_acc_configure() | for the NF to configure the parameters of its desired accelerator module |
| DHL_get_shared_IBQ() | for the NF to get a shared input buffer queue |
| DHL_get_private_OBQ() | for the NF to get its private output buffer queue |
| DHL_send_packets() | for the NF to send the raw data to FPGA |
| DHL_receive_packets() | for the NF to get the processed data from FPGA |

### B. Hardware Function Abstractions

The gist of DHL is that NFs offload the computationally intensive processing to FPGA. However, it is difficult and time-consuming to write or revise code that cover specifics of FPGA offloading and carefully handle vendor-specific details. To solve this, DHL provides *hardware function abstractions* that enable FPGA offloading in the form of a function call.

The hardware function abstractions of DHL provide developers a familiar DPDK programming model that fully encapsulates the low-level specifics of FPGA. The DHL APIs for NFs to interact with FPGA are shown in Table II. Using DHL, a software NF can benefit from FPGA acceleration by easily shifting the software function calls to hardware functions with minimal code changes. In the following, we explain the

```
//pre-processing of packets
rte_ring_dequeue_bulk(ring_in, pkts, n_pkts, NULL);
for (i=0;i<n_pkts;i++) {
  aes_256_ctr(pkts[i], ...);
}
//continue other processing
```

Listing 1. Original software NF code for encryption.

```
//register to DHL and enqueue packets to OBQ
nf_id = DHL_register();
acc_id = DHL_search_by_name(``aes_256_ctr``);
DHL_configure(acc_id, default_conf);
for (i=0; i<n_pkts; i++) {
  pkts[i].nf_id = nf_id;
  pkts[i].acc_id = acc_id;
}
struct rte_ring* IBQ = DHL_get_shared_IBQ(nf_id);
DHL_send_packets(IBQ, pkts, n_pkts);

//fetch packets from the OBQ
struct rte_ring* OBQ = DHL_get_private_OBQ(nf_id);
DHL_receive_packets(OBQ, pkts ,n_pkts);
```

Listing 2. Code with DHL using hardware function calls.

detailed process of transforming a software function call (in Listing 1) to DHL hardware function calls (in Listing 2) using an example of encryption.

From Listing 1, we can see that the NF calls a function called `aes_256_ctr()` to encrypt a set of packets `pkts`. To see how to accelerate the encryption function with DHL, we turn to Listing 2. First the NF needs to register with the DHL Runtime to get its *nf_id*, and searches the *hardware function table* for the hardware function named *aes_256_ctr* to get the *acc_id*. Then it configures the hardware function. Before sending the raw data to FPGA, the *nf_id* and target *acc_id* must be attached to the packets. At last, NF requests a shared input buffer queue (IBQ) from DHL Runtime by specifying its *numa_id* (see Section IV-A2), and enqueues all the packets to the shared IBQ. Meanwhile, the NF needs to poll its private output buffer queue (OBQ) to get packets returned from hardware function.

### C. DHL Runtime

DHL Runtime handles with the underlying FPGA substrate and hides all specifics of them.

In control plane, DHL Runtime manages all the accelerator modules and FPGAs in the system. As Figure 2 shows, the Controller in DHL is key component. On one hand, it responds to the registration request of NFs to assign *nf_id* and create private OBQ for them. On the other hand, it maintains the *hardware function table* and *accelerator module database*. In detail, *hardware function table* stores the mappings of *hf_name* and *acc_id*, and *accelerator module database* stores the PR bitstream of all the supported accelerator modules (more details see Section IV-C).

In data plane, DHL Runtime exploits the lockless ring as the I/O buffer queue to buffer data and provide data isolation among multiple NFs. Moreover, DHL proposes a generic high-throughput low-latency data transfer layer between the host and FPGAs, which exploits several optimization techniques, such as batching data transfer of data packets, using user-space I/O technology to map the register of FPGAs into user-space to bypass the kernel, and polling the DMA engine to further mitigate the latency.

## IV. Implementation

We implement a prototype of DHL [25] on top of DPDK and it needs no modification to DPDK. In this section, we describe the implementation details.

### A. Data Transfer Layer

Since both the latency and throughput are critical to network functions, a low-latency, high-throughput data transfer layer is essential to DHL framework.

*1) UIO-based Poll Mode Driver:* Most of FPGA-based accelerating solutions are designed for large volume data processing (data transfer size usually varies from tens to hundreds megabytes) with millisecond level latency requirement, such as genomic analysis, machine learning, video transcoding and compression. It is unapplicable for network functions due to their ultra-low microsecond-level latency requirement and small packet size that an ethernet packet varies from 64 bytes to 1500 bytes.

To meet the ultra-low latency demand, we design a userspace I/O (UIO) based poll mode driver for the scatter-gather (SG)[4] packet DMA engine for FPGAs. Firstly, a standard in-kernel driver for the peripheral accelerating devices needs to implement a set of standard system interfaces for both data transfer and device configuration. User applications need to use corresponding system calls, like `write()`, `read()`, and `ioctl()`, etc, to interact with the in-kernel driver, which takes many CPU cycles to deal with context switch between user space and kernel space. By utilizing the UIO technology, we map the registers and memory space of FPGAs to the userspace memory address, thus DHL Runtime can directly manipulate the FPGA and bypass the system kernel. Secondly, interrupt is the common means for the peripheral accelerating devices to return the post-processing data. Although it saves CPU cycles to handle other processes, the interrupt handling incurs extra processing latency for content switch. We avoid this by realizing the driver in poll mode that we assign dedicated CPU cores for data transfer. In detail, DHL data transfer layer continually checks whether there is data to be transferred to FPGAs or whether there are data returned from FPGAs.

*2) NUMA Aware:* In a multi-core system, memory access time depends on the memory location relative to a processor (see Figure 3). Similarly, DMA transactions of data transfer layer between FPGA to a remote host memory will degrade I/O performance too. To avoid this, DHL carefully allocates huge pages and sets CPU affinity in a NUMA-aware fashion. The packet descriptors of DMA Engine, shared input buffer

[4] Scatter-gather refers to the process of gathering data from, or scattering data into, the given set of buffers.

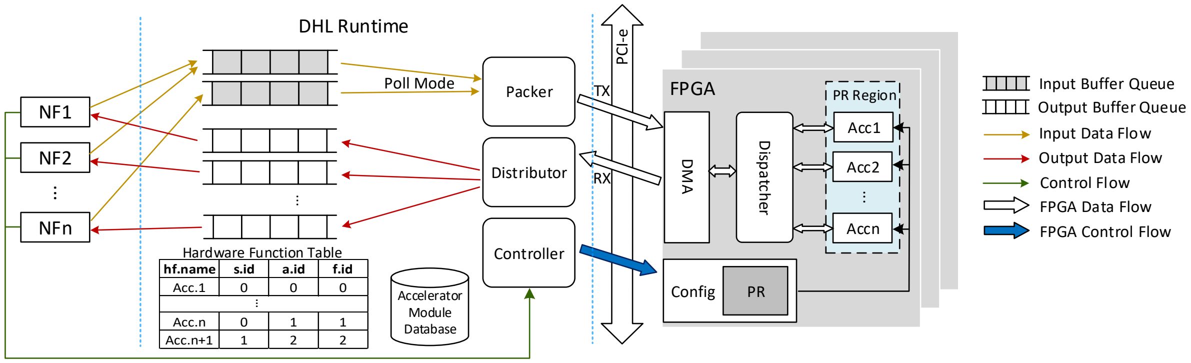


Fig. 2. The detailed architecture of DHL. In *hardware function table*, hf.name: $hf\_name$; s.id: $socket\_id$; a.id: $acc\_id$; f.id: $fpga\_id$.

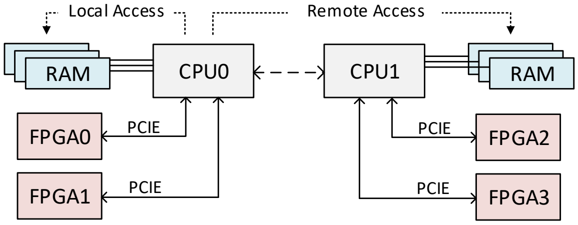
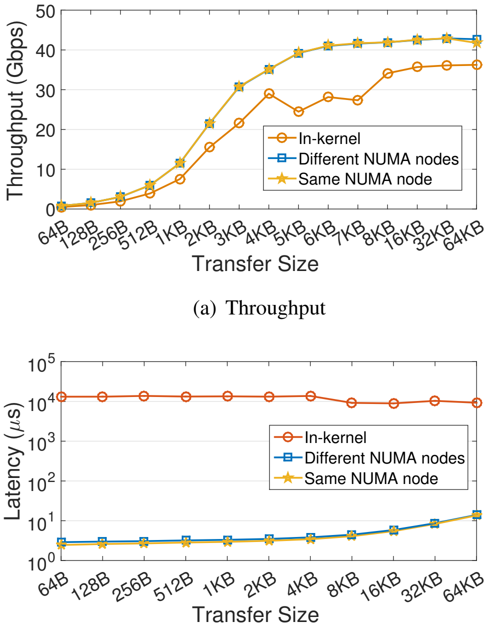


Fig. 3. The block diagram of NUMA architecture server.



(a) Throughput

(b) Latency

Fig. 4. The performance of the packet DMA engine with PCIe GEN3 x8. Same NUMA node means the test is between the local memory and FPGA. Different NUMA node means the test is between a remote memory and FPGA.

queues, and private output buffer queues are allocated in the same NUMA node with the targeted FPGA. Moreover, DHL Runtime schedules the CPU core in the same NUMA node to execute the polling tasks.

*3) Transfer batching:* To evaluate our UIO based poll mode driver and packet DMA engine, we implement a *loopback* module in FPGA that simply redirects the packets received from RX channels to TX channels without any involvement of other components in FPGA. Figure 4 summarizes the performance of our packet DMA engine with different transfer size. As a base-line, we compare our UIO based poll mode driver with the in-kernel driver [26] provided by Northwest Logic, which also runs in poll mode.

Interestingly, shown from Figure 4(a) and Figure 4(b) when evaluating the improvement of NUMA-aware feature, we find that the NUMA-aware memory allocation gains no throughput improvement and only gains about 0.4 $\mu$s (about 800 CPU cycles) latency saving. This indicates that compared to data transfer overhead between host memory and FPGA, the remote memory access overhead is minimal.

Moreover, figure 4(a) shows that our UIO based poll mode driver achieves higher throughput than in-kernel driver with any transfer size. Also as shown in Figure 4(b), the round-trip transfer latency of in-kernel driver is as high as 10 ms, while our UIO based poll mode driver has very low latency of 2 $\mu$s.

However, we can see that up to 42 Gbps throughput, is only obtained for transfer size bigger than 6 KB. Luckily, the latency of 6 KB transfer size is only 3.8 $\mu$s. Since the low throughput of small transfer size may severely bottleneck the ingress traffic of accelerator modules, thus leading a severe accelerating degradation, we aggressively use batching to transfer a bundle of packets to FPGA at a time. On the other hand, the accelerator modules in FPGA usually employ packet level parallelism. This implies the accelerators need to manipulate multiple packets in parallel, and demands the batch transfer of packets between the host and FPGA. To balance the throughput and latency, the maximum batching size is limited at 6 KB.

To achieve batching, there are two components, Packer and Distributor in DHL Runtime (see Figure 2). When DHL Runtime dequeues the shared IBQs to get ingress packets, Packer groups them by $acc\_id$, and encodes the 2-Byte tag pair $(nf\_id, acc\_id)$ into the header of date field. Then the

Packer encapsulates the packets of the same group to perform batching according to the pre-set batching size. At the same time, Distributor polls the FPGAs to get the post-processed packets. It first decapsulates the batched packets, and distribute them to private OBQs in terms of their $nf\_id$.

*4) Shared IBQs & Private OBQs:* As Figure 2 shows, DHL employs the lockless ring, provided by DPDK, to implement the input buffer queues (IBQs, software NFs to data transfer layer) and output buffer queues (OBQs, data transfer layer to software NFs) to buffer and isolate the data between software NFs.

At runtime, DHL needs to check the IBQs to get the raw data and transfer them to FPGAs. Considering that assigning one CPU core to poll all the IBQs increases latency and assigning each IBQ a dedicated CPU core is costly, we treat IBQs and OBQs in different way. Specifically, DHL Runtime creates a multi-producer and single-consumer IBQ for every NUMA node, and each IBQ is shared among the software NFs that runs in the same NUMA node. Differently, DHL Runtime creates a private single-producer and single-consumer OBQ for each software NF so that each software NF can easily get the post-processed data separately.

### *B. Data Isolation*

DHL supports accelerating multiple NFs simultaneously regardless of whether they call the same or different hardware functions. Unlike the software execution model in which CPU interprets the instructions to fetch data from memory to perform processing, the raw data to be processed needs to passes through the hardware circuits like a stream in FPGA. How to ensure the data isolation among multiple accelerator modules and how to distribute the post-processed data to each software NF become a problem. Intuitively, we use the unique identifier to distinguish different NFs and hardware functions.

*1) Distinguish among Multiple NFs:* To distinguish the raw data from different NFs, software NFs need attach their $nf\_id$ to the raw data. When these data returned from FPGA, DHL Runtime can distribute them to different private OBQ based on the $nf\_id$ tag.

*2) Distribute Data to Different Accelerator Modules:* There is a module named Dispatcher in each FPGA, which is responsible for dispatching the data sent from DHL data transfer layer to different accelerator modules based on the $acc\_id$, and re-packing the post-processed data to get them back to host.

### *C. Accelerator Modules & Partial Reconfiguration*

In DHL, software NFs only offload the deep packet processing to FPGA to minimize the programming logic cell consumption of each accelerator module. Therefore, we can put more accelerator modules in the same FPGA to maximize the resource utilization. Further, to meet the demand of the changeable NFV environment and to provide more flexibility, DHL supports reconfigure the accelerator module without interfering other running accelerators on the fly. DHL enables this by adopting the Partial Reconfiguration (PR) technology [27], [28] supported by most of the FPGA vendors.

TABLE III
HARDWARE CONFIGURATION

| Category | Specification |
|---|---|
| CPU | 2x Intel Xeon Silver 4116<br>(12 cores @2.1 GHz, 16.5 MB L3 cache) |
| RAM | 128 GB (DDR4, 2400 MHz, 16GB x 8) |
| NIC | 2x Intel XL710-QDA2 (dual-port 40 GbE)<br>2x Intel X520-DA2 (dual-port 10 GbE) |
| FPGA | Xilinx Virtex-7 VC709 FPGA board<br>( XC7VX690T FPGA, PCIe 3.0 x 8) |

With PR technology, the logics in the FPGA design are divided into two types, static region and reconfigurable region. In DHL, DMA engine, Dispatcher, Config module and PR module belong to static region. Apart from the static region, we divide the rest logic resource into several reconfigurable parts.

When designing an FPGA, we first generate the configuration bitstream of the base design that keeps the reconfigurable parts blank with data and configuration interfaces defined. Then we fill the reconfigurable part with accelerator module (e.g. Encryption, Decryption, MD5 authentication, Regex Classifier, Data Compression, etc) in the base design to generate partial reconfiguration bitstream. These PR bitsteams are stored in the *accelerator module database* managed by the DHL Runtime.

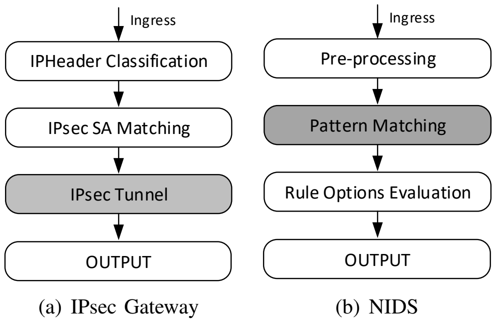
When a software NF looks up the *hardware function table* with its desired $hf\_name$ and $socket\_id$, if there is no matched item in the table, the DHL Runtime will search *accelerator module database* to get the PR bitstream associated with the $hf\_name$ and update the table entry. Then, DHL Runtime calls the $DHL\_load\_PR()$ API to partially reconfigure the FPGA. In detail, DHL Runtime sends the PR bitstream to Reconfig module, and Reconfig module partially reconfigure the target reconfigurable part in FPGA through ICAP (Internal Configuration Access Port) without interfering other running accelerator modules.

Importantly, DHL allows software developers to add their self-built accelerator modules to *accelerator module database* as long as following the specified design specifications. In the base design of FPGA, we apply the same design specifications for every reconfigurable part of a 256 bits width data-path in AXI4-stream protocol and a 250 MHz clock. Following these design specifications, software NF developers can design their customized accelerator modules, and they can generate their customized PR bitstreams by merging them into the base design provided by DHL.

## V. EVALUATION

### *A. Testbed*

First we introduce our evaluation testbed. We use three Dell R740 servers, one equipped with a Xilinx VC709 FPGA board



(a) IPsec Gateway (b) NIDS

Fig. 5. Workflow of IPsec gateway and NIDS.

(with PCIe Gen3 x8) plugged in a PCI-e GEN3 x16 slot on which running the DHL, and the other two acting as a traffic generator. Table III lists the specifications of these servers. All the NICs are connected to a Top-of-Rack (ToR) Dell S6010-ON switch [29]. We use DPDK-Pktgen [30] to generate and sink traffic.

The host OS is Centos 7.4 (kernel 3.10.0), DPDK version is 17.08. We set up 16 GB hugepages for DPDK runtime per server.

### B. Sample Applications

*1) IPsec gateway:* IPsec is widely used to protect communications over Internet Protocol (IP) networks with cryptographic security services. Figure 5(a) shows the generic workflow of IPsec gateway. If the network packets match the IPsec security association (SA)[5], the IPsec gateway performs encryptions and authentication on them. Since cryptographic operations (encryption and authentication) used in IPSec are highly computation-intensive and previous works [8], [14] have shown that software based IPSec gateway behave poor performance, we choose to offload them to FPGA.

*2) Network Intrusion Detection System:* Network intrusion detection system (NIDS) analyzes and monitors the incoming network packets for malicious activity. It usually uses Deep Packet Inspection (DPI) to inspect the entire packet based on the pre-defined signature ruleset. Figure 5(b) shows the generic workflow of an NIDS. The NIDS reads packets, prepares them for pattern matching, runs pattern matching to detect potential attacks, evaluates various rule options, and finally passes or drops the packets based on rule options. The pattern matching used in NIDS is the bottleneck which consumes a lot of CPU cycles for computation [31], [32], we choose to offload it to FPGA.

For evaluation, these two applications are implemented in two versions, CPU-only version and DHL version. The CPU-only version is the pure-software implementation and is built based on the pipeline mode offered by Intel DPDK. In pipeline mode, the application is made up of separate I/O cores and worker cores: the I/O cores are responsible for receiving/sending packet from/to NICs and enqueuing/dequeuing them to/from the rings shared with worker cores, and worker cores are responsible for the asynchronous processing of these packets. In contrast, the DHL version offloads the computation-intensive part (i.e., cryptographic operations of IPsec gateway and pattern matching of NIDS) to FPGA.

We implement an IPsec gateway that encrypts packet payload with the 256-bit AES scheme in CTR mode and creates an authentification digest with HMAC-SHA1. For CPU-only version, we use the Intel-ipsec-mb library[6] for the block encryption and authentication. The Intel-ipsec-mb library uses the Intel Multi-buffer [13] technology along with Intel SSE, AVX, AVX2, AVX512 instruction sets to maximize the throughput that CPU can achieve. In contrast, for DHL version we implement an accelerator module, named *ipsec-crypto*, which is the combination of AES_256_CTR algorithm for encryption and the HMAC_SHA1 algorithm for authentication.

We implement a signature-based NIDS that uses a Snort-based [33] attack ruleset to monitor malicious activity. For CPU-only version, we use Aho-Corasick (AC) pattern matching algorithm [34] to scan ingress network traffic. Correspondingly, we implement an accelerator module named *pattern-matching* by porting the multi-Pattern string matching algorithm [35] for DHL version.

### C. Performance Gains for Single Application

First, we evaluate DHL with only one NF instance that we run IPsec gateway and NIDS separately. We use the Intel XL710-QDA2 NIC which supports up to 40 Gbps traffic and needs 2 I/O CPU cores to achieve this rate. Table IV summaries the configuration of CPU core assignment and batching size. Specifically, we allocate 2 CPU cores for DHL Runtime that one for sending data to FPGA, and the other for receiving data from FPGA, and we set the batching size to 6 KB to match the 40 Gbps line-rate. Also, we allocate 2 CPU cores for workers in CPU-only version. Besides, we design a test that only takes 2 CPU cores to perform the IO operation without computation processing as the base line.

Figure 6 shows the throughput and latency of the IPsec gateway and NIDS. We can see that, taking the same 4 CPU cores DHL version outperforms the CPU-only version. For IPsec gateway, the DHL version reaches close to the peak throughput of I/O throughput, from 19.4 Gbps with 64B packets to 39.6 Gbps with 1500B packets, while the CPU-only version achieves very low throughput, from 2.5 Gbps with 64B packets to 7.3 Gbps with 1500B packets. For NIDS, the DHL version gains the throughput from 18.3 Gbps with 64B packets to 31.1 Gbps with 1500B packets, while the CPU-only version only achieves 2.2 Gbps with 64B packets to 7.7 Gbps with 1500B packets. It is the *pattern-matching* module that limits the maximum throughput of NIDS to 31.1 Gbps. Due to the limited resource of a reconfigurable part, it is restricted

[5] A security association is simply the bundle of algorithms and parameters (such as keys) that is being used to encrypt and authenticate a particular flow in one direction.

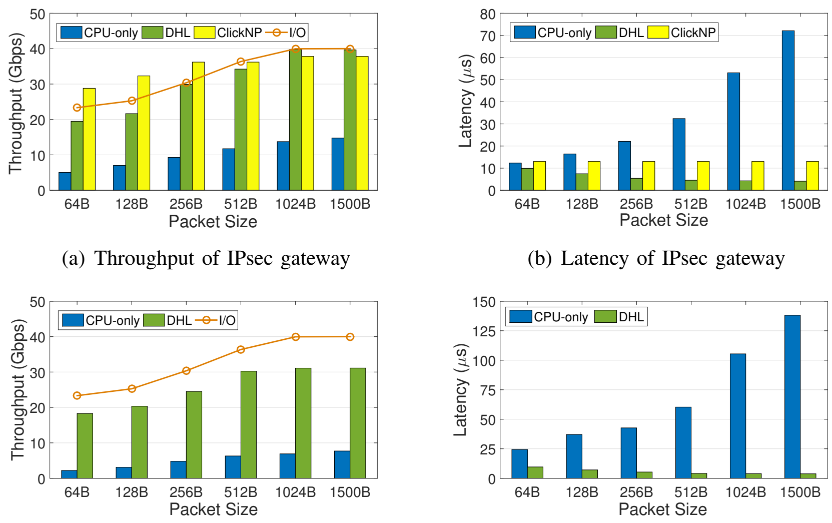
[6] Intel(R) Multi-Buffer Crypto for IPsec Library, a highly-optimized software implementation of the core cryptographic processing for IPsec, which provides industry-leading performance on a range of Intel(R) Processors.

TABLE IV
EXPERIMENT CONFIGURATION

| Test | Software NF | Ethernet I/O | CPU-only | | DHL | | |
|---|---|---|---|---|---|---|---|
| | | | Worker | Total | Runtime | Batching Size | Total cores |
| Single NF | IPsec Gateway | 2 cores | 2 cores | 4 cores | 2 cores | 6 KB | 4 cores |
| | NIDS | 2 cores | 2 cores | 4 cores | 2 cores | 6 KB | 4 cores |
| Multi-NFs with accelerator shared [1] | IPsec Gateway | 1 core | N/A | N/A | 2 cores | 6 KB | 4 cores |
| | IPsec Gateway | 1 core | | | | | |
| Multi-NFs with different accelerators [2] | IPsec Gateway | 1 core | N/A | N/A | 2 cores | 6 KB | 4 cores |
| | NIDS | 1 core | | | | | |

[1] We run two IPsec gateway instances that use the same accelerator module IPsec_crypto.

[2] We run an IPsec gateway instance using the accelerator module IPsec_crypto, and a NIDS instance using the accelerator module pattern_matching simultaneously.



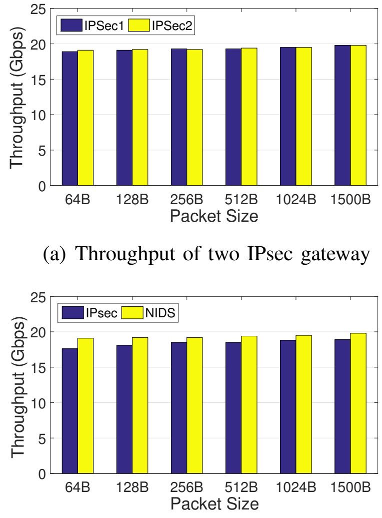
(a) Throughput of IPsec gateway

(b) Latency of IPsec gateway

(c) Throughput of NIDS

(d) Latency of NIDS

Fig. 6. Throughput and processing latency of the single IPsec gateway/NIDS running with 40G NIC.



(a) Throughput of two IPsec gateway

(b) Throughput of IPsec gateway and NIDS

Fig. 7. The throughput of multiple NFs with 10G NICs.

to process no more than 8 characters per clock cycle, which gives a theoretical throughput of 32 Gbps.

We measure the latency by attaching a timestamp to each packet when the NF gets them from NIC. Before the packets are sent out from NIC, we use the current timestamp to minus the saved one. Figure 6(b) and Figure 6(d) shows the processing latency under different load factors. We can see that when packet size increases, the processing latency for CPU-only version increases up to 72 $\mu$s for IPsec gateway and 138 $\mu$s for NIDS. For DHL version, both IPsec gateway and NIDS incurs less than 10 $\mu$s in any packet size. Moreover, we can see that the latency is higher of 64B packets than 1500B packets. Since it needs to encapsulate more packets to reach the batching size of 6 KB for 64B packets, that indeed incurs more time to dequeue enough packets from shared IBQs.

Besides, as ClickNP [14] has implemented an FPGA-only IPsec gateway, we compare our DHL framework with it in terms of IPsec gateway. Since ClickNP is not opensourced that we cannot reproduce their experiment with our platform, we use the result reported in their paper. As shown in Figure 6(a), we can see that the throughput of DHL is close to ClickNP except for small packet size. From Figure 6(b), we find that the IPsec gateway of ClickNP suffers more latency than DHL, it is weird that it should not be so high if based on the delay (cycles) of their elements reported in their paper.

From these single NF tests, we can see that DHL, using four CPU cores, the CPU-FPGA co-design framework, can provide up to 7.7× throughput with less to 19× latency for IPsec gateway and up to 8.3× throughput with less to 36× latency for NIDS .

### *D. Multiple NF Applications*

Since there is only one FPGA board in our testbed, which only can offer up to 42 Gbps data transfer throughput between CPU and FPGA. To evaluate how DHL performs with multiple NFs running simultaneously, it needs to divide the total transfer throughput to each NF instance. So we use two Intel X520-DA2 Dual port NICs (total four 10G ports) to provide four separate 10 Gbps input traffic, total 40 Gbps input traffic, and assign them to different NF instances. Unlike the 40G port of Intel XL710-QDA2 NIC, only one CPU core is enough to

TABLE V
Reconfiguration Time of Accelerator Modules

| Accelerator Module | PR Bitstream Size | PR Time |
|---|---|---|
| *ipsec-crypto* | 5.6 MB | 23 ms |
| *pattern-matching* | 6.8 MB | 35 ms |

saturate the full 10 Gbps at any packet size for 10G port of Intel X520-DA2 NIC.

In detail, we run two IPsec gateway instances to imitate the situation that multiple NFs use the same accelerator module (call the same hardware function), and an IPsec gateway instance along with an NIDS instance to imitate the situation that different NFs use the different accelerator modules in the same FPGA. We assign two 10G ports to each instance and each port assigned with one CPU core for I/O, thus the theoretical maximum throughput of each instance is 20 Gbps in total.

Figure 7(a) shows that both two IPsec gateway using the same accelerator module *ipsec-crypto* can reach the maximum throughput of 20 Gbps with any packets size. However, shown from Figure 7(b), we can see that the throughput of IPsec gateway is a little lower than NIDS. The reason for this is that the *pattern-matching* accelerator modules incurs fewer delay cycles than *ipsec-crypto*, thus NIDS can get the post-processed data returned from FPGA faster.

From this multiple NFs test, we demonstrate that DHL is capable of supporting multiple NFs whether they use the same/different accelerator modules. Furthermore, different accelerator modules may affect the accelerating throughput of NFs a bit due to their different processing delays.

### *E. Partial Reconfiguration*

To evaluate how partial reconfiguration works, we first load the base design into FPGA and remain all the reconfigurable parts blank. Then we start the IPsec gateway by loading the *ipsec-crypto* accelerator module. After the IPsec gateway is stable, we start the NIDS to dynamically reconfigure a free reconfigurable part into *pattern-matching*. We change the launching order and test again. There is no throughput degradation of the running NF when we load the new accelerator module into FPGA.

Table V presents the reconfiguration time of these two accelerator modules. The reconfiguration time is the time interval counted from calling the *DHL_load_pr()* function to load PR bitsteam till status register indicating reconfiguration done. We can see that the reconfiguration time is proportional to the PR bitstream size and is very short.

### *F. Utilization of FPGA*

From Table VI, we can see that both ipsec-crypto and pattern-matching use a small portion of LUTs (Lookup tables), 2.18% for ipsec-crypto and 1.4% for pattern-matching. When it comes to BRAM, the consumption is rather higher than LUTs in which ipsec-crypto consumes 16.46% while pattern-matching consumes 35.64% BRAM. For ipsec-crypto, the reason is our implementation of the cipher operation needs to set up a 28 stages pipeline to provide up to 60 Gbps throughput, which consumes massive registers to hold the intermediate results. For pattern-matching, it is the multiple-pipeline AC-DFA [35] that costs much BRAM. Apart from the static region, there are enough resource to place 5 ipsec-crypto or 2 pattern-matching in an FPGA. If we decrease the size of the AC-DFA pipeline, it can put more pattern-matching accelerator modules. Alternatively, we can use the latest FPGA chip, which contains much more resource.

TABLE VI
Summary of the two accelerator modules and static region

| Accelerator Module | LUTs[1] | BRAM[1] | Throughput (Gbps) | Delay (Cycles) |
|---|---|---|---|---|
| *ipsec-crypto* | 9464 (2.18%) | 242 (16.46%) | 65.27 | 110 |
| *pattern-matching* | 6336 (1.4%) | 524 (35.64%) | 32.40 | 55 |
| Static Region | 136183 (31.43%) | 83 (5.64%) | N/A | N/A |

[1] We use a Xilinx Virtex-7 VX690T FPGA with 433200 LUTs and 1470 (x36Kb) BRAM blocks.

### *G. Developing Effort*

For software development, it is easy to shift the existing software NFs into DHL version by using the DHL programming APIs. As shown from Listing 2 in Section III-B, it just needs minimal modifications (tens of LoC, see Table VII) to turn a pure software implemented NF to DHL version.

TABLE VII
The total lines of code to shift the CPU-only NF into DHL

| Accelerator Module | *ipsec-crypto* | *pattern-matching* |
|---|---|---|
| LoC[1] | 33 | 35 |

[1] Line of code. It represents how many lines of code modified or added to shift a software function call to the hardware function call.

For hardware development, we implement the two accelerator modules, *ipsec-crypto* and *pattern-matching*, by directly writing the verilog code, so that we can fully optimize for performance. Alternatively, it can be quicker to develop the accelerator modules by using the high-level language compilers or tools [14], [21], [22], [36] in the price of some performance degradation.

During the evaluation, DHL eases the developing efforts from two aspects: (i) since DHL decouples the software developing and hardware developing, we do both in parallel, which provides us great flexibility; (ii) after we finished the accelerator modules, we just leave them aside. When adjusting the NFs to conduct different tests, we modify the codes and compile the application totally in software fashion, which significantly saves time.

## VI. Discussion

So far we have demonstrated that DHL is easy to use, and it effectively increases the throughput of software NFs by offloading computation-intensive tasks to FPGA. We discuss related issues, the current limitations, more applicable scenarios here.

*1) Vertical scaling:* As shown in Section IV-A, our prototype can only provide a maximum throughput of 42 Gbps due to the PCI-e 3×8 specification, of which the theoretical bandwidth is 64 Gbps. It restricts the accelerating capacity of an FPGA when the aggregate throughput of NFs exceeds the maximum throughput. To get higher throughput for an FPGA, we can replace with an advanced one (e.g., PCI-e 3×16 with 126 Gbps), alternatively we can install more FPGA cards into the free PCIe slots.

*2) Batching size:* By now, we aggressively set the batching size of DHL Runtime as high as 6KB for maximum DMA throughput. It incurs more waiting time to aggregate packets to reach the batching size for small size packets, thus increasing the latency. For future work, we are going to design a dynamic, adaptive algorithm that DHL Runtime adjusts the batching based on the traffic size. When the traffic is small, it decreases the batching size to reduce latency.

*3) Applicable to more:* In this paper, we demonstrate the ability of DHL to accelerate software NFs with the sample applications of IPsec gateway and NIDS. Since DHL provides a set of transparent APIs for software developers to interact with FPGA, it seems naturally applicable to other software applications that contain computation-intensive tasks. However, to support more (e.g., machine learning), it needs to modify the data structure and DMA engine. Since DHL targets software NFs, we choose the unified packet structure of DPDK – $rte\_mbuf$, which is highly optimized for networking packets, and has a limited maximum data size for 64 KB. Moreover, DHL uses the scatter-gather DMA engine, it is also highly optimized for networking packets which is usually small.

## VII. Related Work

Click [37] offers a modular software architecture for router implementation so as to enable easy programmability and scalability. However, software NFs suffer lower performance than hardware appliance. RouteBricks [6] is a scalable software router that parallelizes packet processing with multiple CPU cores as well as scaling across a cluster of servers to fully exploit the CPU cores. It achieves 35 Gbps packet forwarding performance with four servers. DoubleClick [38] improves the performance by employing batching to packet I/O and computation, showing 10x improvement to reach 28 Gbps for IPv4/IPv6 packet forwarding rate.

To accelerate software packet processing, there is much work exploiting GPUs. PacketShader [8] demonstrates the feasibility of 40 Gbps packet forwarding speed for 64B packets with GPU as accelerators. However, to achieve the 40 Gbps, it requires up to 1024 batching size, which incurs at least 200 $\mu$s latency. To avoid the overhead of PCIe communication of GPU, APUNet [39] proposes a APU-based packet processing platform which decreases the maximum latency to 3.4 $\mu$s.

Due to the re-programmability and the ability to customize the hardware, there is a large number of work implementing network functions with FPGA, such as load balancer [20], DNS server [21], IPsec gateway [14], Network Intrusion Prevention [40]. They all demonstrate that FPGA can provide better performance than software solutions with higher throughput and lower latency. However, most of these works put the entire network function in FPGA, thus lacks flexibility and scalability as we discussed in this paper.

We are not the first to combine CPU and FPGA to accelerate network functions. For example, VNRE [17] gives an example of using FPGA to accelerate NRE (Network Redundancy Elimination), which offloads Rabin fingerprint computation and CDC algorithm to FPGA. ClickNP [14] claims that it supports joint CPU/FPGA processing and shows examples of traffic generator and logger. However, there is no work providing a unified framework for software network functions. To the best of our knowledge, DHL is the first general framework for CPU-FPGA co-design.

Designing FPGA needs to use HDLs, making it difficult for the software developers who lack hardware skills. There are some work providing developing tools and techniques to simplify FPGA programming. ClickNP [14] proposes a modular design and utilizes high-level synthesize tools to develop network functions in FPGA with high-level language. Emu [21] is a framework that enables application developers to write network services in a C# language and makes them automatically compiled to FPGA platform. We can fully utilize these tools to help rapidly develop and test accelerator modules.

## VIII. Conclusion

In this paper, we first review current FPGA-based solutions to accelerate network functions. Almost all of current solutions tend to implement the entire NF on FPGA, which exposes several drawbacks (e.g., resource-wasting and costly, long compilation time), leading to high cost and lack of programming flexibility to meet varying requirements in real-world deployment. To tackle the above shortcomings, we propose DHL, a CPU-FPGA co-design framework for high performance packet processing. DHL allows the main body of NFs (e.g., packets I/O and control logic) to run on CPU and offloads computation-intensive processing to FPGA for acceleration. It decouples the software-level programming and hardware-level acceleration, and provides abstractions which not only keeps the flexible programming for software NFs modification, but also enables high performance of NFs with less FPGA resources. In addition, it supports multiple software NF applications to call the same/different hardware functions on FPGA simultaneously without interfering each other. Finally, we build a prototype of DHL system on a server equipped with a Xilinx FPGA. Our evaluation demonstrates that DHL can achieve similar throughput (as high as 40 Gbps) using much fewer FPGA resources, while incurring reasonable latency less than 10 $\mu$s.

## References

[1] Q. Zhang, Y. Xiao, F. Liu, J. C. S. Lui, J. Guo, and T. Wang, "Joint optimization of chain placement and request scheduling for network function virtualization," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 731–741.

[2] T. Wang, H. Xu, and F. Liu, "Multi-resource load balancing for virtual network functions," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 1322–1332.

[3] X. Fei, F. Liu, H. Xu, and H. Jin, "Towards load-balanced vnf assignment in geo-distributed nfv infrastructure," in *Quality of Service (IWQoS), 2017 IEEE/ACM 25th International Symposium on*. IEEE/ACM, 2017, pp. 1–10.

[4] ——, "Adaptive vnf scaling and flow routing with proactive demand prediction," in *INFOCOM 2018 - The 37th Annual IEEE International Conference on Computer Communications*. IEEE, 2018, pp. 1–9.

[5] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2014, pp. 459–473.

[6] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: exploiting parallelism to scale software routers," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 15–28.

[7] L. Rizzo, "Netmap: A novel framework for fast packet I/O," in *USENIX Annual Technical Conference*, 2012.

[8] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," in *Proceedings of the ACM SIGCOMM 2010 conference*. ACM, 2010, pp. 195–206.

[9] Z. Xu, F. Liu, T. Wang, and H. Xu, "Demystifying the energy efficiency of network function virtualization," in *Quality of Service (IWQoS), 2016 IEEE/ACM 24th International Symposium on*. IEEE, 2016, pp. 1–10.

[10] C. Zeng, F. Liu, S. Chen, W. Jiang, and M. Li, "Demystifying the performance interference of co-located virtual network functions," in *INFOCOM 2018 - The 37th Annual IEEE International Conference on Computer Communications*. IEEE, 2018, pp. 1–9.

[11] I. Corporation, "Intel data plane development kit," http://dpdk.org/.

[12] "DPDK Intel NIC Performance Report," http://fast.dpdk.org/doc/perf/DPDK_17_05_Intel_NIC_performance_report.pdf.

[13] Intel, "Processing multiple buffers in parallel to increase performance on intel architecture processors."

[14] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, and P. Cheng, "Clicknp: Highly flexible and high-performance network processing with reconfigurable hardware," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 1–14.

[15] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "Netfpga sume: Toward 100 gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, 2014.

[16] M. B. Anwer and N. Feamster, "Building a fast, virtualized data plane with programmable hardware," in *Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*. ACM, 2009, pp. 1–8.

[17] X. Ge, Y. Liu, C. Lu, J. Diehl, D. H. Du, L. Zhang, and J. Chen, "Vnre: Flexible and efficient acceleration for network redundancy elimination," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 83–92.

[18] "LiquidIO II." http://www.cavium.com/pdfFiles/LiquidIO_II_CN78XX_Product_Brief-Rev1.pdf.

[19] D. Unnikrishnan, R. Vadlamani, Y. Liao, A. Dwaraki, J. Crenne, L. Gao, and R. Tessier, "Scalable network virtualization using fpgas," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2010, pp. 219–228.

[20] S. Atalla, A. Bianco, R. Birke, and L. Giraudo, "Netfpga-based load balancer for a multi-stage router architecture," in *Computer Applications and Information Systems (WCCAIS), 2014 World Congress on*. IEEE, 2014, pp. 1–6.

[21] N. Sultana, S. Galea, D. Greaves, M. Wojcik, J. Shipton, R. Clegg, L. Mai, P. Bressana, R. Soulé, R. Mortier, P. Costa, P. Pietzuch, J. Crowcroft, A. W. Moore, and N. Zilberman, "Emu: Rapid prototyping of networking services," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 459–471. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/sultana

[22] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4FPGA: A Rapid Prototyping Framework for P4," in *Proc. SOSR*, 2017.

[23] T. N. Thinh, T. T. Hieu, S. Kittitornkun *et al.*, "A fpga-based deep packet inspection engine for network intrusion detection system," in *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2012 9th International Conference on*. IEEE, 2012, pp. 1–4.

[24] O. Elkeelany, M. M. Matalgah, K. P. Sheikh, M. Thaker, G. Chaudhry, D. Medhi, and J. Qaddour, "Performance analysis of ipsec protocol: encryption and authentication," in *Communications, 2002. ICC 2002. IEEE International Conference on*, vol. 2. IEEE, 2002, pp. 1164–1168.

[25] "OpenCloudNeXt DHL." https://github.com/OpenCloudNeXt/DHL.

[26] N. Logic, "Northwest logic dma driver," http://nwlogic.com/products/docs/DMA_Driver.pdf.

[27] Xilinx, "Vivado design suite user guide: Partial reconfiguration (ug909)," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug909-vivado-partial-reconfiguration.pdf.

[28] Intel, "Introduction of altera partial reconfiguration," https://www.altera.com/products/design-software/fpga-design/quartus-prime/features/partial-reconfiguration.html.

[29] "Dell networking s6000 spec sheet." http://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/dell-networking-s6100-on-specsheet.pdf.

[30] "DPDK-Pktgen." http://dpdk.org/browse/apps/pktgen-dpdk/.

[31] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, "Kargus: a highly-scalable software-based intrusion detection system," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 317–328.

[32] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon, "Nba (network balancing act): A high-performance packet processing framework for heterogeneous processors," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 22.

[33] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks." in *Lisa*, vol. 99, no. 1, 1999, pp. 229–238.

[34] M. Alicherry, M. Muthuprasanna, and V. Kumar, "High speed pattern matching for network ids/ips," in *Network Protocols, 2006. ICNP'06. Proceedings of the 2006 14th IEEE International Conference on*. IEEE, 2006, pp. 187–196.

[35] W. Jiang, Y.-H. E. Yang, and V. K. Prasanna, "Scalable multi-pipeline architecture for high performance multi-pattern string matching," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–12.

[36] S. Singh and D. J. Greaves, "Kiwi: Synthesis of fpga circuits from parallel programs," in *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on*. IEEE, 2008, pp. 3–12.

[37] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM SIGOPS Operating Systems Review*, vol. 33, no. 5, pp. 217–231, 1999.

[38] J. Kim, S. Huh, K. Jang, K. Park, and S. Moon, "The power of batching in the click modular router," in *Proceedings of the Asia-Pacific Workshop on Systems*. ACM, 2012, p. 14.

[39] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park, "Apunet: Revitalizing gpu as packet processing accelerator." in *NSDI*, 2017, pp. 83–96.

[40] N. Weaver, V. Paxson, and J. M. Gonzalez, "The shunt: an fpga-based accelerator for network intrusion prevention," in *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*. ACM, 2007, pp. 199–206.