Parking Packet Payload with P4

Swati Goswami, Nodir Kodirov, Craig Mustard, Ivan Beschastnikh, Margo Seltzer
University of British Columbia
sggoswam@cs.ubc.ca,knodir@cs.ubc.ca,craigm@ece.ubc.ca,bestchai@cs.ubc.ca,mseltzer@cs.ubc.ca

ABSTRACT

Network Function (NF) deployments suffer from poor link goodput, because popular NFs such as firewalls process only packet headers while receiving and transmitting complete packets. As a result, unnecessary packet payloads needlessly consume link bandwidth. We introduce PayloadPark, which improves goodput by temporarily parking packet payloads in the stateful memory of dataplane programmable switches. PayloadPark forwards only packet headers to NF servers, thereby saving bandwidth between the switch and the NF server. PayloadPark is a transparent in-network optimization that complements existing approaches for optimizing NF performance on end-hosts.

We prototyped PayloadPark on a Barefoot Tofino ASIC using the P4 language. Our prototype, when deployed on a top-of-rack switch, can service up to 8 NF servers using less than 40% of the on-chip memory resources. The prototype improves goodput by 10-26% for a $Firewall \rightarrow NAT$ NF chain and reduces PCIe bandwidth load by 2-58%. With workloads that have datacenter network traffic characteristics, PayloadPark provides a 13% goodput gain with a $Firewall \rightarrow NAT \rightarrow LB$ NF chain, without latency penalty. In this scenario, we can further increase the goodput gain to 28% by using packet recirculation.

CCS CONCEPTS

• **Networks** → **Programmable networks**; **Middle boxes / network appliances**.

KEYWORDS

Programmable switches, Network function virtualization, P4

ACM Reference Format:
Swati Goswami, Nodir Kodirov, Craig Mustard, Ivan Beschastnikh, Margo Seltzer. 2020. Parking Packet Payload with P4. In *The 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20), December 1–4, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3386367.3431295

1 INTRODUCTION

Network Functions (NFs) are widely deployed in enterprise networks, and they implement critical network functionality, such as intrusion detection, Network Address Translation (NAT), and performance optimization (WAN optimizers, caches) [32]. NFs are often connected together in an *NF chain* [15] and deployed using software modules running on commodity hardware. However, lack of specialized hardware results in degraded performance [9, 12]. Moving NFs to the cloud further degrades performance due to additional network hops and encryption overhead [32].

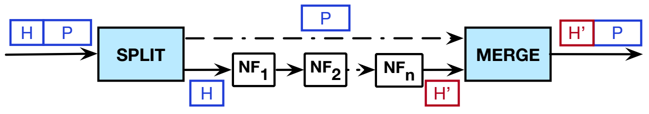

Figure 1: Abstract PayloadPark deployment. *Split* decouples the packet into header H and payload P. Shallow NF chain NF_1, NF_2 ... NF_n transforms header H into H'. *Merge* reassembles the header H' with payload P.

NFs that process only headers, such as firewalls and NATs, suffer from *poor goodput*, because unexamined packet payloads waste link bandwidth. *Goodput* is the amount of useful information delivered over time. In this case, goodput is the amount of data examined by the NF. For example, NATs and firewalls can achieve *throughput* that saturates a 40 Gbps link when processing 10 Mpps with 500 byte (4000 bits) packets. But, they only examine the 5-tuple in the first 42 bytes (336 bits) of the header (including Ethernet, IPv4, and UDP header). So their *goodput* is only 3.36 Gbps. To increase goodput, we propose PayloadPark – a header-payload decoupling optimization that temporarily holds packet payloads in switch dataplane memory. PayloadPark forwards only packet headers to NFs, temporarily parks payloads in the switch ASIC memory, and reassembles the packet when returned by the NFs.

As PayloadPark addresses bandwidth, it is orthogonal to prior work that optimizes endhosts, such as reducing per-packet CPU cycles (SpeedyBox [16], NetBricks [28]). In fact, PayloadPark complements such existing end-host optimizations. For example, a combined setup of PayloadPark and NetBricks results in goodput improvement from PayloadPark and throughput and latency gains from NetBricks. Also, NF frameworks that use specialized devices such as FPGAs [20] or programmable NICs [19] are bandwidth bound and will yield a higher benefit by improving the packet processing rate – a result of improved link goodput.

PayloadPark is appropriate for header-only and fixed-prefix NFs, i.e., *shallow* NFs, such as NATs, firewalls, and L4 load balancers and not services that require deep packet inspection, e.g., intrusion detection. A prior survey shows that, on average, 44% of datacenter traffic requires at least one of L4 load-balancing and NAT operations [29]. Moreover, with increasingly encrypted traffic [25], many NFs are effectively limited to shallow processing and will benefit from PayloadPark.

Fig. 1 shows an abstract deployment of PayloadPark and its two primitive operations, *Split* and *Merge*. *Split* decouples the incoming packet's header, H, from its payload, P. Header H is forwarded to the shallow NF chain $NF_1, NF_2...NF_n$, while payload P is parked

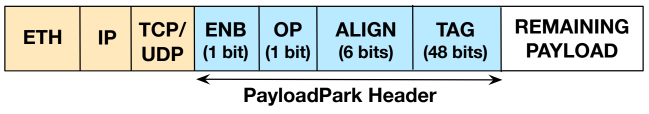

Figure 2: PayloadPark header. Enable (ENB) bit indicates whether PayloadPark operation is enabled. Opcode (OP) indicates the operation to be performed: Merge | Explicit Drop. ALIGN bits are for byte-alignment. TAG is a unique identifier for the packet.

(stored) in the switch dataplane. *Merge* re-combines the (potentially modified) header H' from the NF chain with payload P before forwarding the packet to its destination. In our prototype, we deploy Split and Merge on the same ToR switch. This improves goodput, because the link between the switch and the NF servers transfers only useful information (headers) and not unused data (payload).

While intuitive, PayloadPark has only recently become possible thanks to newly available Reconfigurable Match-Action Table (RMT) switches [4]. RMT switches are equipped with programmable ASICs that create opportunities for implementing in-network optimizations using domain specific languages such as P4 [3]. RMT switches have limited storage resources and impose limits on the number of per-packet compute and stateful operations. These restrictions ensure packet processing at line rate, but make PayloadPark implementation challenging. *PayloadPark fits within these limits by 1) relying on the low-latency nature of NFs to bound required storage so it can evict old payloads, and 2) by turning the optimization off when switch memory is exhausted.*

Prior work, such as Dejavu [34] and SilkRoad [24], move functionality onto the switch (e.g., offloading entire chains or specific functionality, such as load balancing). Such deployments provide high performance but limit flexibility in NF implementation. SilkRoad stores active connection state in the switch, so it suffers performance degradation when the number of active connections exceed stateful memory resources on the switch [23]. Our work differs from these approaches in that PayloadPark is a *transparent in-network optimization* that leaves NF chains to run on commodity hardware. This retains the *flexibility* in implementing NF chains and ensures ease of integration with existing NF frameworks such as OpenNetVM [35] and NetBricks [28].

2 BACKGROUND: RMT SWITCHES

RMT switches process packets using *user-defined* headers. This flexibility enables PayloadPark to process packet payloads in the switch dataplane. At a high level, RMT switches work by passing each packet through a series of match-action tables (MATs), which perform *actions* (functions) on packet headers that *match* criteria.

More precisely, the packet processing pipeline is composed of three building blocks: Parser, Match-Action Pipeline, and Deparser. The *Parser* interprets the packet header using a user-defined header and populates the Packet Header Vector (PHV). The PHV size is vendor dependent, and it limits the overall size of packet headers that can be processed in the switch dataplane. The *Match-Action Pipeline* is composed of stages, where each stage has local ALU, SRAM, and TCAM resources. The match-action pipeline is programmed by writing match-action table (MAT) definitions in P4 [3]. Each MAT contains several entries composed of *match* and *action* rules. For each packet, a MAT compares header fields (from the PHV) using user-supplied *match* rules and executes an *action* based on the comparison result. MATs can also perform stateful operations using the read/write register API. The register API abstracts SRAM as an array of fixed-sized bit-vectors called *registers*. To ensure line-rate packet processing, switches impose restrictions on the number of per-packet stateful operations. MATs communicate values, such as register contents, to subsequent stages using user-defined metadata values that are allocated on the PHV. Finally, the *Deparser* assembles the new header from the modified and unmodified header fields.

The packet processing pipeline can recirculate the packet by sending the packet back to the parser. Recirculation increases the number of permitted per-packet header transformations but results in a bandwidth and latency penalty.

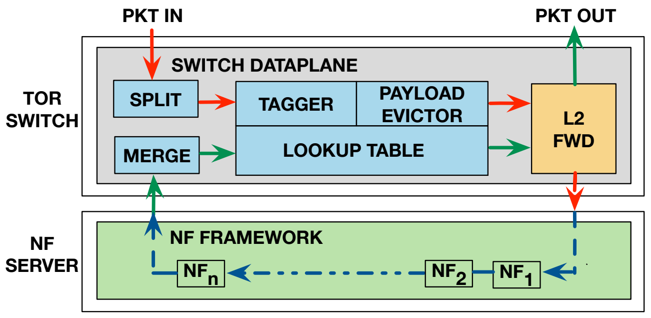

Figure 3: Packet flow in PayloadPark. *Split* decouples the packet header and payload, and stores the payloads in the lookup table. The *Merge* operation merges the headers from the NF server with the payloads stored in the lookup table. L2 FWD forwards packet using L2 forwarding.

3 PAYLOADPARK OVERVIEW

3.1 PayloadPark Header

PayloadPark is enabled on a per-port basis. When a packet arrives on a PayloadPark-enabled port, the Split operation adds a PayloadPark header (shown in Fig. 2). The Enable (ENB) bit indicates if the payload was successfully stored in the switch. The opcode (OP) bit distinguishes between Merge and Explicit Drop operation (discussed in §5.2). The tag is a unique identifier for each packet, and it is used to map the packet to the payload in the switch dataplane.

3.2 High Level Algorithm

PayloadPark implements two operations: *Split* and *Merge*:

- **Split** decouples the packet header and payload, by (1) associating a unique tag with the packet, (2) storing the payload in the dataplane, (3) adding the PayloadPark header to the packet, and (4) setting the Enable bit to one (ENB bit in Fig. 2). The Split operation sets the Enable bit to zero when there is insufficient memory to store the payload.
- **Merge** recombines the payload with the modified header from the NF chain. The Merge operation (1) uses the tag to locate the stored payload, (2) appends the payload to the packet, (3) removes

the PayloadPark header, and (4) frees the space consumed by the payload.

The Split and Merge functionality is composed of three components: the packet tagger, lookup table, and payload evictor (Fig. 3).

Packet tagger. Every packet must be assigned a unique identifier or *tag* that is used to index into the lookup table. The original packet header cannot be used for indexing, because NF chains can modify headers.

Lookup table. PayloadPark uses a lookup table to store packet payloads. The lookup table is composed of two tables – the metadata and payload tables – each organized as register arrays indexed using a common table index. The metadata table is conceptually a bitmask indicating which positions in the payload table are occupied.

Payload evictor. The evictor reclaims memory occupied by payloads of packets lost or dropped after the Split operation. Packet drops may occur due to lossy links or when packets are dropped by NFs, such as firewalls. The NF framework will not notify the switch of packet drops, because the framework is unaware of PayloadPark operation on the switch. For eviction, we associate every entry in the metadata table with an expiry threshold (EXP). When an entry reaches its expiry, the evictor reclaims space in the lookup table. The Merge operation distinguishes between evicted and non-evicted payloads using the packet tag. PayloadPark works within limited storage by giving the NF enough time to return a packet before the payload is evicted.

The switch uses L2 forwarding to send packets to their destination, and the NF framework processes the packets through an NF chain. L2 forwarding and the NF framework run independently of the PayloadPark components.

4 SWITCH DATAPLANE DESIGN

We next map the general-purpose design described in §3.2 onto the architecture of the switch dataplane. The code is available on Github [1], and our companion paper [11] includes a pseudocode.

Split operation. When the switch receives a packet, it executes the Split operation. In the first stage, we maintain two counters (each two bytes wide) to generate two parts of the tag: 1) an index to find an empty location for storing the packet payload (TI in Fig. 4), and 2) a generation number to disambiguate between evicted and non-evicted payloads (CLK in Fig. 4). We increment these counters and roll them over when they reach their maximum values.

In stage 2, we probe the metadata table using the table index to determine if the index is empty. Each entry in the metadata table includes the expiry threshold (EXP), and the index is available if EXP is zero. If the index is available, we claim it for storing payload by writing the CLK value and a pre-defined expiry threshold (EXP) into the metadata table. For example, in Fig. 4, assume that when the Split operation started, the EXP value in element 0 (the TI) was 0. We then write CLK and EXP into the zeroth entry of the metadata table. We add the PayloadPark header to the packet and update the ENB and tag values. If the lookup table entry at the table index is occupied, we set the ENB bit to zero. We also add a 2 byte CRC. It is a part of the tag and is used to validate the PayloadPark header.

If we found an empty location in stage 2, then we store the packet payload in subsequent stages. The payload table is a two dimensional array, where the columns are spread across MATs. To match this memory layout, we stripe the incoming payload into equal-sized blocks, called *payload blocks*. Following the example in Fig. 4, we store the payload at the zeroth row by striping the payload blocks, $P_0, P_1 ... P_L$, across all the columns. The total number of payload bytes stored in the payload table will vary across switches; it depends on PHV size and available memory resources. **In our implementation, we store 160 bytes of payload per packet**. We do not split packets with payload size less than 160 bytes to avoid wasting dataplane memory resources. For small packets (payload size < 160 bytes), we add the PayloadPark header and set the ENB bit to zero.

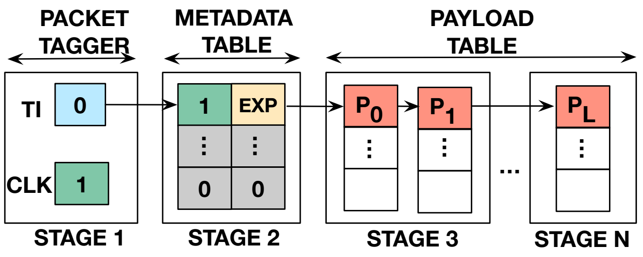

Figure 4: PayloadPark dataplane implementation. The tagger has registers for the table index (TI), which is an index into the lookup table, and a clock (CLK). The metadata table contains two values at each index, the value of the clock when the index was occupied and an expiry threshold (EXP). If the index is available for storing payload, its EXP value is 0. Payload Blocks (P_0, P_1 ... P_L) are striped across MATs.

The Split operation does a single lookup in the metadata table to find empty slots due to restrictions imposed by the switch. Laying out stateful memory as arrays, or more accurately, a circular buffer, neatly coexists with this restriction. Usually, Split and Merge process packets in the same (FIFO) order. Thus, if we allocate space in the metadata table sequentially, as the table index works its way through the array, Merge operations reclaim memory at earlier positions in the array. By the time the table index wraps around, it should find empty spots. This access pattern optimizes PayloadPark operation for the common case (FIFO order).

Payload eviction. The Split operation also reclaims memory in the lookup table by cleaning up long-lived payloads. Each entry in the metadata table stores the entry's expiry threshold. If during the Split operation, the TI points to an occupied location (indicated by a non-zero value of the expiry threshold), we decrement the expiry threshold. When the associated expiry threshold reaches zero, we evict the stored payload and reclaim the space for splitting packets. For example, if value of EXP is 2, the TI will iterate over the metadata table twice, before evicting the payload.

Merge operation. When the switch receives a packet from the NF server, we execute the Merge operation. In the first stage, we process packets for which PayloadPark operation was disabled. For such packets, we remove the PayloadPark header and forward packets to their destination.

In the second stage, we validate that the packet's stored payload has not been evicted by comparing the clock values in the PayloadPark header and the metadata table. If the validation succeeds, we

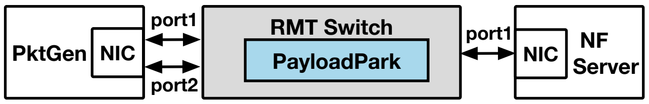

Figure 5: Experimental setup.

reclaim the space in the metadata table and remove the PayloadPark header. If validation fails, we conclude that the payload was prematurely evicted. We drop the packet, and update the premature eviction counter.

In subsequent stages, we merge the payload back to the validated packets and remove the stored payload from the lookup table.

5 EVALUATION

We compare PayloadPark's performance with a baseline and show that PayloadPark uses switch resources efficiently.

Setup. We implemented PayloadPark using approximately 900 lines of $P4_{16}$ [6] code and deployed it on a 64 port 6.4 Tbps switch with Barefoot Tofino ASIC [26]. The switch has four pipes where 16 ports share the resources of each pipe. For the baseline setup, the switch uses L2 forwarding to pass traffic between the traffic generator and the NF server.

We use two NF frameworks – OpenNetVM [35] and NetBricks [28] to evaluate our prototype. OpenNetVM is built on top of DPDK and runs NF chains in Docker containers [35]. NetBricks is a DPDK-based framework written in Rust, which provides software-level isolation. We evaluated PayloadPark using Intel 82599ES 10 GE NIC and Intel XL710 40 GE NIC.

We use PktGen, a DPDK-based traffic generator to saturate the NF server with UDP packets. We run PktGen on a dual NUMA node, 2.4 GHz Intel Xeon E5-2407 v2 server with 8 cores and 48GB RAM. Fig. 5 shows the experimental setup. We connect two ports of the PktGen's NIC to the switch to saturate the NF server. One port is not sufficient, because PayloadPark reduces data that the switch transmits to the NF server. We use identical NICs for the traffic generator and the NF server.

The NF server is a four NUMA node, 60 core machine with a 2.3 GHz Intel Xeon E7-4870 v2 processor and 512GB RAM. With OpenNetVM, we reserve 3 cores for the OpenNetVM manager, and each NF in the chain is pinned to one core. With NetBricks, we use 4 cores to run the NF chain. We can scale up the NF framework by using additional cores or ports, but this tuning is orthogonal to our evaluation, because PayloadPark is an in-network optimization. We reserve 8GB of memory backed by hugepages on each NUMA node. We use two NF chains: *Firewall* $\rightarrow$ *NAT* $\rightarrow$ *LB* and *Firewall* $\rightarrow$ *NAT* to evaluate our prototype. The firewall linearly probes through a list of blocked IP addresses. The firewall in the three-NF chain has 20 rules, and the two-NF chain has a single rule in its firewall. The load balancer is based on the Maglev load-balancer [8]. The NAT is based on MazuNAT from NetBricks [28].

Evaluation metrics. PayloadPark is a *goodput* optimization, and we measure goodput from the switch's perspective. We use the packet header as the unit of useful information. In our evaluation, throughput of 10 Mpps corresponds to 3.36 Gbps of goodput in the baseline and PayloadPark, because the packet header (including Ethernet, IPv4 and UDP header) length is 42 bytes (336 bits). We

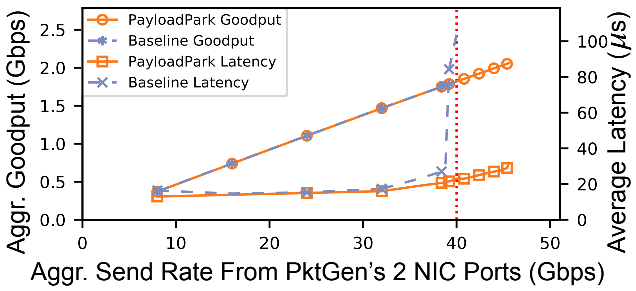

Figure 6: Results for *FW* $\rightarrow$ *NAT* using PktGen. The vertical red line at X=40 Gbps highlights physical link capacity. For X<40 Gbps, the maximum difference between peak and average latency is 31 μs and 25 μs for baseline and PayloadPark. At X=40 Gbps, this difference is 104 μs and 25 μs for baseline and PayloadPark, respectively.

measured end-to-end packet processing latency from the traffic generator. We also measured PCIe bus utilization on the NF server using Intel's Processor Counter Monitor [10]. We consider the system to be healthy when the packet drop rate is below 0.1%; we use this boundary to measure peak goodput of PayloadPark and the baseline. Unless explicitly stated, the expiry threshold (EXP) is set to 1, and *the number of premature evictions is zero* – a prerequisite for functional equivalence between PayloadPark and baseline operation.

Workload. We replay PCAP files to simulate an enterprise datacenter traffic pattern reported by Benson et al. [2]. The packet sizes have a bimodal distribution, with an average packet size of 882 bytes (refer Fig. 8(a) in the appendix). Recall that we do not split packets whose payloads have fewer than 160 bytes. In this workload, 30% of the packets are less than 160 bytes and for such packets, we add the PayloadPark header and set the ENB bit to zero. We collect performance metrics by replaying packets over a period of 2 minutes. For all experiments, we report averages of three runs. Although we use UDP for our evaluation, PayloadPark works with any protocol.

5.1 Performance and Ease of Integration

5.1.1 Performance improvement with replayed PCAP files. Fig. 6 shows the goodput and latency of PayloadPark with the OpenNetVM framework. We used the *FW* $\rightarrow$ *NAT* chain with the 40 GE NIC. We observed similar results with NetBricks, but omit the results for brevity. The experiment shows that PayloadPark can process more traffic than the baseline without latency penalty. PayloadPark integrates easily with NF frameworks, so running on the two different frameworks required no code changes. We also evaluated the *FW* $\rightarrow$ *NAT* $\rightarrow$ *LB* chain with a 10 GE NIC using OpenNetVM framework. **With the 10 GE NIC, we observe a 13.06% goodput improvement and no latency penalty**. The goodput gain in this setup is capped by the NIC capacity.

Fig. 6 also shows that PayloadPark reaches the latency cliff at a higher packet send rate (X>40 Gbps). In the baseline setup, the average latency increases sharply as the network link approaches

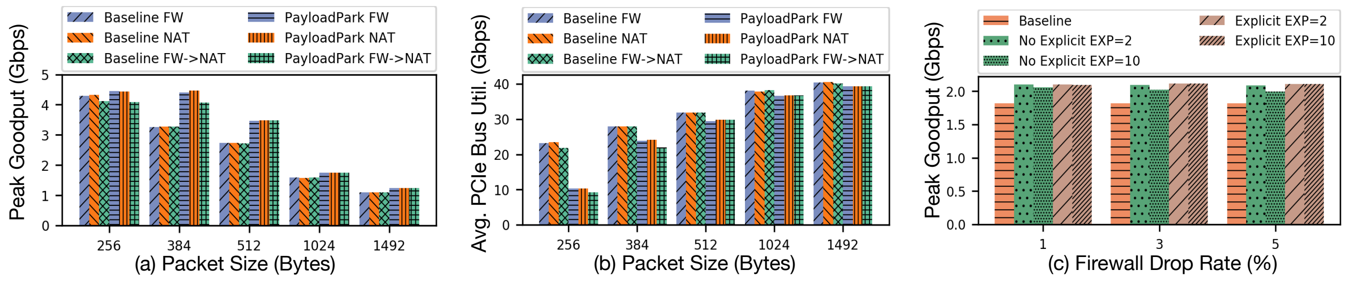

Figure 7: Evaluation with 40 GE NIC (a) (Higher is better) Goodput with different packet sizes, (b) (Lower is better) PCIe bus utilization with different packet sizes, (c) (Higher is better) Effect of expiry threshold and eviction policy with FW->NAT chain.

saturation (X=40 Gbps), while PayloadPark has no such spike, because the switch-to-NF server link does not approach saturation.

The bimodal packet distribution is representative of datacenter traffic, but it yields modest goodput gain with PayloadPark. The reason is two-fold: 1) 30% of the traffic is not Split because packet size is less than 160 bytes, and 2) the average packet size is 882 bytes, so we don't truncate a substantial proportion of the payload.

5.1.2 Goodput Gain with Different Packet Sizes. Fig. 7(a) and Fig. 7(b) show PayloadPark's behavior with different packet sizes and NF chains with the 40 GE NIC and OpenNetVM framework. **With PayloadPark, goodput improvement varies between 10% and 36% for 384- to 1492-byte packets.** We see a larger goodput gain for small packets (384 and 512 byte packets) than large packets (1024 and 1492 byte packets), because we truncate a larger fraction of each packet for small packet sizes. Also, the NF framework is able to sustain the higher incoming packet rate for packets larger than 384 bytes. The *FW* $\longrightarrow$ *NAT* chain has lower goodput gain than individual NFs because the NF server does more per-packet computation, which makes OpenNetVM compute bound sooner. For 256 byte packets, the goodput gain is negligible, because the NF framework becomes compute bound and is not able to sustain the higher packet rate in the PayloadPark deployment.

Fig. 7(b) shows a 2 - 58% reduction in PCIe bus load; the reduction is proportional to the number of payload bytes stored in the switch. Neugebauer et al. show that PCIe bandwidth decreases with small packets [27]. However, PayloadPark still provides a net goodput gain. See further discussion in our companion paper [11]. Assuming that future generations of RMT switches have more memory, the PCIe savings will increase, because we can store more payload bytes on the switch. Another option is to deploy PayloadPark on SmartNICs. However, this approach will not improve goodput on the link. Also, prior work showed that NFs can run directly on SmartNICs [20, 22]. NF deployment on SmartNICs can simultaneously benefit from performance gains of SmartNICs and goodput gains from PayloadPark operating on the switch.

5.2 Effect of Expiry Threshold

We added 50 lines of code to OpenNetVM to explicitly notify the switch when the NF drops a packet. Explicit Drop notifications provide ground truth to evaluate the payload eviction policy, because the NF notifies the switch as soon as a payload can be evicted. OpenNetVM marks a packet as dropped by changing the opcode ("OP" field in Fig. 2), truncating the packet payload, and sending the resulting (header-only) packet back to the switch. The Drop operation reclaims switch memory after validating the tag.

Resource Name	Percentage Res. Util.
SRAM (4 NF servers)	25.94% (Avg.) / 33.75% (Peak)
TCAM	0.69%
Exact Match Crossbar	16.47%
Ternary Match Crossbar	0.88%
Packet Header Vector (PHV)	37.65%

Table 1: Resource utilization on the Tofino chip.

Fig. 7(c) compares the goodput with and without explicit drops to different expiry thresholds. We replay our PCAP files with the 40 GE NIC for the *Firewall* $\longrightarrow$ *NAT* chain running on OpenNetVM. We vary the proportion of blocked IP addresses in the FW to control the drop rate at the firewall. Unlike the firewall benchmark recommendations [14], we consider dropped packets in our measurement, since we measure goodput from the switch's perspective.

Fig. 7(c) shows that an aggressive eviction policy (EXP=2) performs comparably to Explicit Drop notifications. With a conservative eviction policy (EXP=10), goodput reduces, because dropped packets stay longer and occupy more space in the lookup table.

Overall, the expiry threshold trades effective memory utilization for protection against premature payload evictions. Explicit Drops *in combination* with payload eviction balances this trade-off. Fig. 7(c) shows that a conservative eviction policy with Explicit drops (Explicit EXP=10) performs comparably to an aggressive eviction policy (No Explicit EXP=2). This benefit comes at a one-time cost of small code changes to the NF framework.

5.3 Resource Utilization

Payload Park is designed to take advantage of the spare capacity already available in Tofino switches. Table 1 shows the dataplane resources used by our prototype (excluding L2 forwarding). Despite being memory-intensive, our average per-stage SRAM utilization is 25.94%, and it is comparable to prior work (SilkRoad [24], BurstRadar [18]). This memory is sufficient for supporting four NF servers (one on each pipe). PHV resource utilization of 37.65% is comparable to our overall memory consumption and therefore, not a limiting resource. Additional resources such as TCAM and crossbars also have less than 20% utilization. Overall, PayloadPark leaves sufficient resources for implementing additional P4 functionality.

6 DISCUSSION

Limitations. PayloadPark increases packets per second on the links, but the overall performance gain is capped by the performance profile of the end-host NF server. If the NF server setup cannot process the incoming packet rate in the baseline setup, the overall setup will not benefit from PayloadPark, because additional packets that PayloadPark transfers over the link will get dropped at the NF server. NF frameworks that use commodity hardware are compute bound and this limits the goodput gain. This is reflected in our choice of simple NFs and shorter NFs for evaluation with the 40 GE NIC. With the 10 GE NIC, goodput gain is capped by the NIC capacity. Also, longer packet processing latency (longer NF chain or slower NF framework) increase the memory pressure on the switch, limiting goodput gain with PayloadPark. In the worst case, when switch memory is exhausted, PayloadPark gracefully stops parking payload bytes on the switch and adds a fixed PayloadPark header overhead (of 7 bytes) per packet.

Improving goodput gain. Our evaluation shows that storing only 160 bytes on the ToR switch can be effective. We can further increase the goodput gain by recirculating packets within the switch (see Appendix B). We can also split packets across switches in the datacenter topology: core, aggregate and the ToR switch. We can increase the goodput gain and distribute memory pressure by striping the packet payload across multiple switches in the packet path within the cloud provider's infrastructure.

Adaptive payload eviction policy. Our prototype tracks premature payload evictions with a counter. We use this counter to identify the safe operating boundary of PayloadPark. This counter could be used to adaptively change the payload eviction policy and protect against unexpected latency spikes in the NF server. For example, PayloadPark could start with an aggressive payload eviction policy and dynamically switch to a conservative eviction policy when payload evictions exceed a predefined threshold. We leave this tuning to future work.

Failure scenarios. The PayloadPark deployment and the baseline setup deal with the following failure scenarios:

1) Link/NF server failures: Both deployments are equally susceptible to NF server failures and link disconnects. All in-flight messages in disconnected link(s) and the server will be lost. Incoming traffic will trigger payload eviction, but overall NF processing will stall. Once the disconnected links and NF server are restored, the payload evictor will reclaim space and resume normal operation.

2) Switch failure: PayloadPark increases the failure-domain of the NF-server to include the switch. Therefore, when the switch fails, all packets in the switch will be lost in both deployments. However, the failure scenario differs for packets that have already reached the NF server. In the baseline deployment, if the server connects to another ToR switch, it can route packets through a different ToR. But, this is not applicable to PayloadPark due to the stored payloads in the (failed) switch. However, the number of extra lost packets is negligible due to the small time-delta between Split and Merge operation. For example, in Fig. 6, PayloadPark will lose at most 200 packets (compared to the baseline) for a link operating at full capacity (40 Gbps) with average packet size of 882 bytes and average latency of 32 μs. With a 10 Gbps link, we will lose at most 50 packets. This packet loss is an overestimate, because we measure latency from the traffic generator and not from the switch.

7 RELATED WORK

Our work is inspired from Cut Payload proposed by Cheng et al. [5]. Cut Payload drops payload at overloaded switches to accelerate TCP congestion detection. PayloadPark makes a similar observation about not transmitting unnecessary data. Our approach is different in that we do not drop the payload. Instead, we store the payload at the switch and later merge it back with the header. In addition, our work spans two areas: in-network computing and NFs.

In-network computing. Prior work accelerates applications by offloading application functionality to programmable switches. PayloadPark differs by using the switch to implement a transparent optimization. NetCache implements an in-network cache for key-value stores [17]. NetPaxos [7] offloads some Paxos functions to the switch. NOPaxos (Network Ordered Paxos) uses in-network devices for network sequencing and accelerates data replication [21].

Specialized hardware for NFs. Prior work has used specialized hardware to improve NF performance. For example, ClickNP [20] uses FPGAs and PacketShader [13] uses GPUs for accelerating NFs. Metron offloads stateless NF operations to the switch and programmable NICs, resulting in better latency and throughput [19].

End-host optimizations for NFs. CoMB consolidates NFs to reduce provisioning and maintenance costs [31]. SafeBricks protects NFs in untrusted clouds, but at a performance cost [30]. NFP [33] and Parabox [36] explore parallel paths in execution of NF chains to reduce packet processing latency. These optimizations are orthogonal and PayloadPark can be integrated with such frameworks.

8 CONCLUSION

We described PayloadPark, an in-network optimization that decouples packets at the header-payload boundary. PayloadPark improves shallow NF goodput by 2-36% without any latency penalty, reduces NF server's PCIe load by 2-58%, and uses less than 40% of Tofino chip resources. Also, PayloadPark preserves the semantics of non-PayloadPark deployments, and can be easily integrated with existing NF frameworks.

9 ACKNOWLEDGEMENTS

We thank the anonymous reviewers and shepherd for their insightful and constructive feedback. We would also like to thank Bruce Shepherd, Joel Nider, and William Anthony Mason for reviewing early drafts of the paper and giving us valuable feedback. This research is supported by Natural Sciences and Engineering Research Council of Canada (NSERC) under grant number 2014-04870.

REFERENCES

[1] 2020. PayloadPark on Github. https://github.com/PayloadPark/payloadpark
[2] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC '10)*. ACM, 267–280. http://doi.acm.org/10.1145/1879141.1879175
[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (2014), 87–95. http://doi.acm.org/10.1145/2656877.2656890

[4] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proceedings of the 2013 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '13)*. ACM, 99–110. http://doi.acm.org/10.1145/2486001.2486011

[5] Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. 2014. Catch the Whole Lot in an Action: Rapid Precise Packet Loss Notification in Data Centers. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14)*. USENIX Association, 17–28. http://dl.acm.org/citation.cfm?id=2616448.2616451

[6] The P4 Language Consortium. 2017. P4 16 Language Specification. Retrieved Oct. 30, 2019 from https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf

[7] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. NetPaxos: Consensus at Network Speed. In *Proceedings of the Symposium on SDN Research (SOSR '15)*. ACM, 5:1–5:7. http://doi.acm.org/10.1145/2774993.2774999

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

[9] ETSI. 2013. NFV Whitepaper. Retrieved Oct. 30, 2019 from portal.etsi.org/NFV/NFV_White_Paper2.pdf

[10] The Linux Foundation. 2011. Process Counter Monitor. Retrieved Oct. 30, 2019 from https://github.com/opcm/pcm

[11] Swati Goswami, Nodir Kodirov, Craig Mustard, Ivan Beschastnikh, and Margo Seltzer. 2020. Parking Packet Payload with P4. arXiv:cs.NI/2006.05182 https://arxiv.org/abs/2006.05182

[12] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. 2015. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine* 53, 2 (2015), 90–97. https://doi.org/10.1109/MCOM.2015.7045396

[13] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: A GPU-accelerated Software Router. In *Proceedings of the 2010 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '10)*. ACM, 195–206. http://doi.acm.org/10.1145/1851182.1851207

[14] IETF. 2003. Benchmarking Methodology for Firewall Performance. Retrieved Oct. 30, 2019 from https://tools.ietf.org/html/rfc3511

[15] IETF. 2003. Service Function Chaining Use Cases In Data Centers. Retrieved Oct. 30, 2019 from https://tools.ietf.org/html/draft-ietf-sfc-dc-use-cases-06

[16] Yimin Jiang, Yong Cui, Wenfei Wu, Zhe Xu, Jiahan Gu, K. K. Ramakrishnan, Yongchao He, and Xuehai Qian. 2019. SpeedyBox: Low-Latency NFV Service Chains with Cross-NF Runtime Consolidation. In *International Conference on Distributed Computing Systems (ICDCS '19)*. IEEE, 68–79. https://doi.org/10.1109/ICDCS.2019.00016

[17] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, 121–136. http://doi.acm.org/10.1145/3132747.3132764

[18] Raj Joshi, Ting Qu, Mun Choon Chan, Ben Leong, and Boon Thau Loo. 2018. BurstRadar: Practical Real-time Microburst Monitoring for Datacenter Networks. In *Proceedings of the 9th Asia-Pacific Workshop on Systems (APSys '18)*. ACM, 8:1–8:8. http://doi.acm.org/10.1145/3265723.3265731

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

[20] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '16)*. ACM, 1–14. http://doi.acm.org/10.1145/2934872.2934897

[21] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, 467–483. http://dl.acm.org/citation.cfm?id=3026877.3026914

[22] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19)*. USENIX Association, USA, 363–378.

[23] James McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. 2019. Thoughts on Load Distribution and the Role of Programmable Switches. *SIGCOMM Comput. Commun. Rev.* 49, 1 (Feb. 2019), 18–23. https://doi.org/10.1145/3314212.3314216

[24] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, 15–28. http://doi.acm.org/10.1145/3098822.3098824

[25] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. 2014. The Cost of the "S" in HTTPS. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT '14)*. ACM, 133–140. http://doi.acm.org/10.1145/2674005.2674991

[26] Barefoot Networks. 2019. Tofino ASIC. Retrieved Oct. 30, 2019 from https://www.barefootnetworks.com/products/brief-tofino/

[27] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. ACM, 327–341. https://doi.org/10.1145/3230543.3230560

[28] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, 203–216. http://dl.acm.org/citation.cfm?id=3026877.3026894

[29] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. 2013. Ananta: Cloud Scale Load Balancing. In *Proceedings of the 2013 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '13)*. ACM, 207–218. https://doi.org/10.1145/2486001.2486026

[30] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2018. SafeBricks: Shielding Network Functions in the Cloud. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*. USENIX Association, 201–216. https://www.usenix.org/conference/nsdi18/presentation/poddar

[31] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. 2012. Design and Implementation of a Consolidated Middlebox Architecture. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*. USENIX Association, 24–24. http://dl.acm.org/citation.cfm?id=2228298.2228331

[32] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. 2012. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *Proceedings of the 2012 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '12)*. ACM, 13–24. http://doi.acm.org/10.1145/2342356.2342359

[33] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. 2017. NFP: Enabling Network Function Parallelism in NFV. In *Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, 43–56. http://doi.acm.org/10.1145/3098822.3098826

[34] Dingming Wu, Ang Chen, T. S. Eugene Ng, Guohui Wang, and Haiyong Wang. 2019. Accelerated Service Chaining on a Single Switch ASIC. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets '19)*. ACM, 141–149. https://doi.org/10.1145/3365609.3365849

[35] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. 2016. OpenNetVM: A Platform for High Performance Network Service Chains. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox '16)*. ACM, 26–31. http://doi.acm.org/10.1145/2940147.2940155

[36] Yang Zhang, Bilal Anwer, Vijay Gopalakrishnan, Bo Han, Joshua Reich, Aman Shaikh, and Zhi-Li Zhang. 2017. ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. ACM, 143–149. http://doi.acm.org/10.1145/3050220.3050236

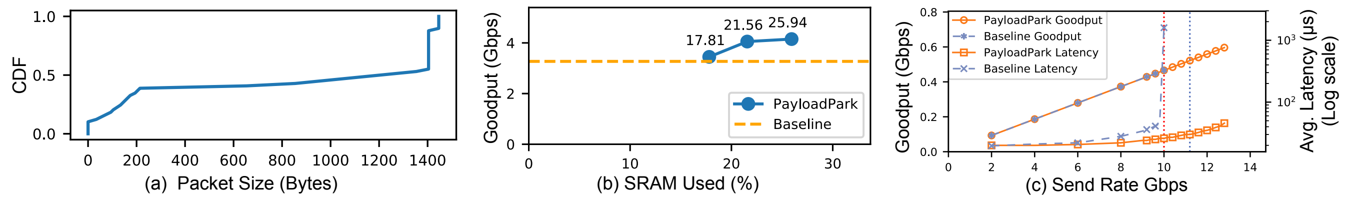

Figure 8: (a) Packet size CDF for replaying PCAP, (b) (Higher is better) Effect of % reserved memory on goodput gain with FW->NAT chain and 40 GE NIC, (c) Goodput and latency with packet recirculation with FW->NAT chain and 10 GE NIC. The vertical red (X=10 Gbps) and blue (X ≈ 11 Gbps) lines highlight maximum send rate that baseline and PayloadPark can sustain without recirculation.

A IMPACT OF SWITCH MEMORY

The amount of memory that PayloadPark reserves on the switch presents a trade-off between goodput gain and memory available for implementing additional P4 functionality. We use OpenNetVM framework with the *Firewall* $\longrightarrow$ *NAT* chain and a workload of 384 byte packets to stress the memory resources at the switch. We set the expiry threshold to 1 and increase the traffic rate until PayloadPark begins to evict packets prematurely. Premature payload evictions must be zero to ensure functional equivalence between PayloadPark and baseline deployment. We test with EXP=1 because if there are no premature payload evictions with an aggressive payload eviction policy (EXP=1), the system will be functionally equivalent with more conservative (EXP>1) payload eviction policies.

Fig. 8(b) shows the peak traffic send rate that exhibits no premature payload evictions with different switch memory allocations. *First, we see that the PayloadPark optimization provides goodput gain with less than 26% Tofino chip memory.* Additionally, with a little more than 17% of switch memory resources, the prototype can sustain at most 3.44 Gbps goodput. Beyond this rate, the expiry threshold is not high enough to prevent premature evictions. For example, at an incoming traffic goodput rate of 3.55 Gbps (send rate of 32.45 Gbps), we observed that 0.03% of incoming payloads are being prematurely evicted (not shown in figure).

B EFFECT OF PACKET RECIRCULATION

In our prototype, we increase the number of stored payload bytes from 160 bytes to 352 bytes by recirculating packets in the packet processing pipeline. Recall that we store payload blocks by striping them across Stages 3 to N of a single pipe (see Fig. 4). Using packet recirculation, we stripe payloads in all the stages of a second pipe, in addition to the payload blocks stored in the first pipe.

Fig. 8(c) shows the goodput and latency with *FW* $\longrightarrow$ *NAT* $\longrightarrow$ *LB* using NetBricks framework with 10GE NIC. The vertical red and blue lines highlight the peak send rate that the baseline and our prototype can sustain (without recirculation). We observe 28% goodput improvement – approximately 2x improvement over the prototype without recirculation. A single packet recirculation induces latency penalty of the order of 10s of ns [34]. But, we do not observe any end-to-end latency penalty thanks to the reduced PCIe latency caused by the additional payload bytes stored in the switch. Without recirculation, we observe 12% reduction in PCIe bus load at all send rates before baseline link gets saturated. With recirculation, NF server's PCIe bus load deceases by 23% in comparison to the baseline. With these results, we conclude that goodput improves and PCIe load decreases with an increase in the number of payload bytes stored in the switch.

C MULTIPLE NF SERVERS AND FUNCTIONAL EQUIVALENCE

This section examines how PayloadPark benefits multiple NF servers, since performance isolation is important in multi-tenant clouds. We simulate such a setup by connecting the switch to 8 NF servers (two on each pipe on the Tofino chip). We increase the reserved memory resources on the switch to about 40% and slice the memory equally amongst NF servers. Each NF server runs on an 8 core, dual NUMA node, 2.4 GHz Intel Xeon E5-2407 v2 machine. Each NF server runs a MAC address swapper using OpenNetVM and services traffic with 384 byte packets. We used small packets, because for a fixed link throughput, they exert more memory pressure on the switch than larger packets. **All 8 NF servers exhibit consistent performance improvement with an average goodput gain of 31.22% and latency win of 9.4%.** We have omitted the graphs for brevity. The latency savings are on the PCIe bus, because PayloadPark copies less data between the NIC and the CPU. This experiment also shows that PayloadPark efficiently uses the on-chip memory resources, because it can service traffic for multiple NF servers. Static slicing of memory resources ensures performance isolation and protects payloads from being evicted by other customer's traffic.

We also validated functional equivalence by comparing the packets upon return from the NF server in the PayloadPark and baseline deployments. The packets captured using DPDK-pdump are identical, and switch metrics report no premature payload evictions.