NFSlicer: Data Movement Optimization for Shallow Network Functions

Anirudh Sarma Hamed Seyedroudbari Harshit Gupta Umakishore Ramachandran Alexandros Daglis
Georgia Institute of Technology

arXiv:2203.02585v1 [cs.NI] 4 Mar 2022

***Abstract*—Network Function (NF) deployments on commodity servers have become ubiquitous in datacenters and enterprise settings. Many commonly used NFs such as firewalls, load balancers and NATs are shallow—i.e., they only examine the packet's header, despite the entire packet being transferred on and off the server. As a result, the gap between moved and inspected data when handling large packets exceeds $20\times$. At modern network rates, such excess data movement is detrimental to performance, hurting both the average and 90% tail latency of large packets by up to $1.7\times$. Our thorough performance analysis identifies high contention on the NIC-server PCIe interface and in the server's memory hierarchy as the main bottlenecks.**

We introduce NFSlicer, a data movement optimization implemented as a NIC extension to mitigate the bottlenecks stemming from data movement deluge in deployments of shallow NFs on commodity servers. NFSlicer only transfers the small portion of each packet that the deployed NFs actually inspect, by *slicing* the packet's payload and temporarily storing it in on-NIC memory. When the server later transmits the processed packet, NFSlicer *splices* it to its previously sliced payload. We develop a software-based emulation platform and demonstrate that NFSlicer effectively minimizes data movement between the NIC and the server, bridging the latency gap between small and large packet NF processing. On a range of shallow NFs handling 1518B packets, NFSlicer reduces average and 90% tail latency by up to 17% / 29%, respectively.

I. Introduction

Network Functions (NFs) have been transitioning from specialized middleboxes to virtualized counterparts on commodity servers [25], [28], [37], [50], which increases their flexibility and facilitates their deployment, boosting their ubiquity in enterprise and datacenter settings [48]. The vast majority of internet traffic that targets an online service is first filtered through several NFs deployed on general-purpose servers, offering functionality such as load balancing within the datacenter, and applying forwarding and firewall rules. Due to the latency-sensitive nature of modern online services, the latency incurred on each packet's NF processing is an important figure of merit.

NFs as a workload are extremely network-bandwidth intensive, as they typically involve minimal processing per received packet. Modern high-bandwidth NICs are therefore a welcome addition to NF-handling servers, as they enable better resource consolidation in the datacenter. However, a side-effect of increasing line rates is unprecedented network traffic moving on and off the server, promoting data movement to a first-order performance determinant for latency-sensitive NFs.

Fig. 1 exemplifies the impact of network data movement, by showing the end-to-end response latency distribution for

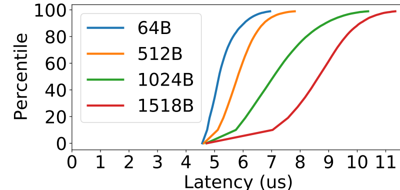


Fig. 1: L2 forwarder NF latency CDF for variable size packets at 7Mpps arrival rate, measured on directly connected client-server machines equipped with 100G NIC (details in §V).

an L2 forwarder NF handling network traffic of four different packet sizes. While the base latency for all packet sizes is the same (∼4.5µs), the latency gap quickly grows as a function of packet size to $1.7\times$ between 1518B and 64B packets. Although the latency difference may initially seem intuitive, L2 forwarders, like *many NFs used in production, are shallow*—i.e., they only operate on packet headers, hence processing cost is insensitive to packet size. We therefore posit that the observed latency gap is primarily attributed to redundant data movement. Given the deluge of large-packet network traffic—video will account for 82% of internet traffic by 2022 [7]—mitigating inefficiencies in its handling is critical.

In this work, we propose NFSlicer, a data movement optimization implemented as a NIC extension to mitigate the latency overheads encountered by shallow NF when handling large packets. NFSlicer is a software-hardware co-design that minimizes data movement between a server executing shallow NFs and its NIC, by only transferring the small subset of the packet the NFs need. The core of NFSlicer is a new basic on-NIC operation, called packet *Slice & Splice*: as each network packet bound to be processed by a shallow NF arrives from the network, NFSlicer *slices* the packet's payload, temporarily stores it in on-NIC memory resources, and extends the packet's header with special metadata. After a CPU on the server executes the NF of the received header and transmits it back to the network, NFSlicer uses these metadata to locate the outgoing packet's corresponding payload and *splices* it back to the header to reconstruct the full packet before placing it on the wire. Slice & Splice addresses large-packet handling inefficiency at its source, shrinking server-NIC data transfers by more than $20\times$ for large packets.

The basic idea of payload slicing for shallow NFs was recently introduced by Goswami et al.'s PayloadPark [15]. Although NFSlicer's design bears strong similarity with PayloadPark, our contributions are distinct. PayloadPark employs

a *partial* Slice & Splice approach on a programmable switch to improve link goodput. In contrast, NFSlicer demonstrates, for the first time, (i) the system-level effects of redundant on-server network data movement; and (ii) the performance gain potential that is *only* attainable by eliminating it with *full* payload slicing, which is beyond the capabilities of switch-based solutions. We further show that NFSlicer's NIC-based mechanism is fundamentally more scalable than switch-based approaches. Overall, while PayloadPark and NFSlicer share significant conceptual similarities, they are complementary mechanisms with distinct strengths.

In summary, we make the following contributions:

- We conduct a thorough microarchitectural study and identify that the noticeable latency gap between large- and small-packet NF processing is attributed to redundant data movement. Our analysis identifies the primary bottleneck on the PCIe interface, and on memory bandwidth to a lesser extent. Thus, the only solution to bridge that latency gap is to reduce the amount of data moved.
- We design the protocol and hardware extensions necessary to realize NFSlicer as a NIC extension.
- We develop a software emulation platform to evaluate NFSlicer's performance improvement potential for a range of shallow NFs. Under the throughput limitations of our emulation platform, we show that for MTU-size (i.e., 1500B) packets, NFSlicer improves the median and 90% tail latency of shallow NFs by 17–20% and 9–29%, respectively. We further show that for higher packet rates, the tail latency improvement potential grows to 55%.
- We synthesize a hardware implementation of NFSlicer to quantify the resource needs and added latency of the Slice & Splice operation, demonstrating NFSlicer's feasibility.

The rest of the paper is organized as follows. §II provides brief background on NFs, typical network packet sizes, and modern NIC capabilities. §III and §IV describe NFSlicer's design and our corresponding emulation platform implementation, respectively. We detail our methodology in §V and evaluate NFSlicer in §VI. We present a hardware implementation and our synthesis results in §VII. Finally, §VIII discusses limitations and potential extensions, §IX covers related work, and §X concludes.

II. Background

A. Network Functions and Modern NF Deployment

NFs broadly range from network architecture controllers implementing SDN control functionality to simple networking utilities such as software switches, routers, Network Address Translators (NAT), firewalls, intrusion detection systems, load balancers, WAN optimizers and flow monitors [33]. Several of these NFs such as firewalls, load balancers, NAT, and switches are "shallow"—i.e., they do not require the entire packet for processing, as they only inspect and modify the packet's L2–L4 headers. NFSlicer's data movement optimization is directly applicable to all such shallow NFs.

Historically, NFs were usually deployed on specialized middleboxes. However, the recent trend of NF virtualization has triggered a major shift from such middleboxes to commodity off-the-shelf servers. The reason for this transition is not performance, but rather ease of deployment and improved resource consolidation in multi-tenant cluster environments.

B. Large-Packet Dominance of Internet Traffic

NFSlicer is an optimization specifically targeting large network packets, which dominate Internet traffic. A large contributor to this trend is the growing demand for video, in the form of IPTV, media streaming, surveillance, etc. To illustrate, video accounted for 58% of total internet traffic in 2018 [49] and is expected to grow to 82% by 2022 [7]. We empirically confirm the prevalence of large-packet network traffic by crunching a packet capture from an Internet backbone link provided by the Center for Applied Internet Data Analysis [4]. We find that 49% of all packets are at least 1400B and 57% are larger than 500B. More importantly, large packets completely dominate the data volume moved on the Internet: 84% / 93% of total data volume is attributed to packets ≥1400B / ≥500B, respectively. As most of this traffic is routed through several NFs deployed on commodity servers, NFSlicer's optimization for large-packet NF processing has broad applicability.

C. Advanced NICs

In the last few years, we have witnessed an explosion in NIC capabilities, both in terms of added functionality and/or programmability. It is now common for modern NICs to offer specialized hardware for networking functionality offload, such as checksum computations and encryption. As *what* we should be accelerating on NICs is still unclear, a second growing trend is the provisioning of programmable hardware resources on the NIC [26], [35], [39], [45].

Two technological trends are contributing to the evolution of NICs in these directions. First, the end of Moore's Law is pushing architects toward hardware specialization, which is fueling an appetite for in-network computing [47]. Architects have found renewed interest in the quest for networking and application functionality that can be moved from the CPU to specialized hardware residing in network gear, such as switches and NICs. Second, increasing line rates (commercial NICs can already drive 2x200G of network traffic [41]) require more pins, which in turn require larger dies and more available on-NIC silicon that can be leveraged to offer additional functionality. Given the momentum for NIC evolution and specialization, NFSlicer is an appealing NIC extension promising significant latency gains for an important workload class that can be realistically deployed in production environments in the near future. We consider a hardened IP block as the most fitting implementation of NFSlicer's Splice & Splice operation, as it is a basic primitive that can be leveraged across shallow NFs. However, in the near term, a soft form of the operation can also be implemented on a NIC with programmable hardware resources.

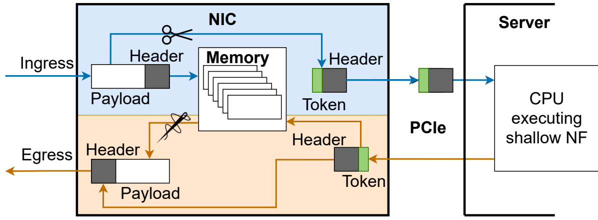


Fig. 2: NFSlicer overview.

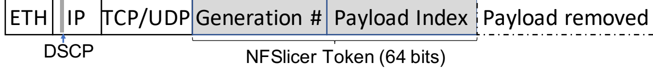


Fig. 3: Packet layout after being processed by NFSlicer.

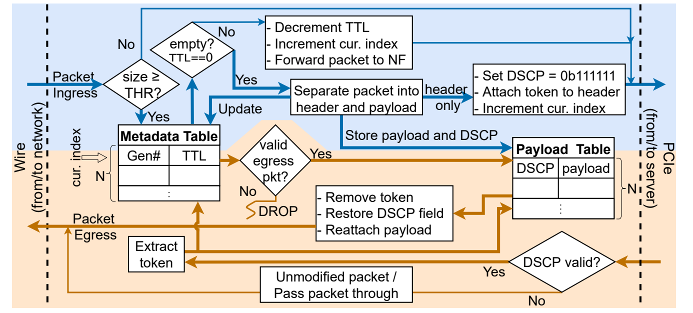


Fig. 4: Slice & Splice pipeline. Top half: slice op; bottom half: splice op. Thick lines highlight the common path.

III. NFSlicer Design

We begin this section with NFSlicer's design overview, elaborate on the software-hardware interface to enable packet slicing, and then describe the Slice & Splice pipeline.

A. Overview

NFSlicer is a NIC extension designed to avoid moving a packet's payload between a server and its NIC, by temporarily storing it in on-NIC memory resources. Fig. 2 displays a high-level view of NFSlicer's operation. As a full packet arrives from the network, the NIC *slices* its *entire* payload and stores it in local memory. The payload is replaced by a small *token*, which serves as a unique identifier to retrieve the sliced payload later, on the packet's egress path. The transformed packet is then transferred to the server, where the server applies its shallow NF processing and transmits it back to the NIC. On the packet's egress, NFSlicer uses the embedded token to retrieve the packet's corresponding payload and *splices* it to the processed header, before transmitting the reconstructed packet on the wire. As pointed out in §I, the design of NFSlicer's mechanism is similar to Goswami et al.'s PayloadPark [15], with the following key differences: (i) NFSlicer targets a NIC-based implementation rather than a switch-based implementation that has to conform to fundamental limitations associated with RMT (Reconfigurable Match Table) pipelines; and (ii) instead of *partial* payload slicing to improve network link goodput, NFSlicer enables *entire* payload slicing to eliminate data movement bottlenecks between the NIC and the server.

B. Software-Hardware Interface

Fig. 3 shows NFSlicer's software-hardware interface. We use the IP header's DSCP field, by setting it to a special value to mark packets that have been sliced. We use the value $0b111111$, which is reserved for experimental/local use [23]. When NFSlicer has available resources to slice and buffer an incoming packet's payload, it sets the DSCP special value, removes the packet's payload, and extends its header with the *NFSlicer token,* a 64-bit value that uniquely identifies a sliced packet's corresponding payload. The token is later used on the packet's egress path—after it has been processed by the NF on the server—to retrieve the packet's corresponding payload for reconstruction. §III-C details the purpose of the token's two fields—*payload index* and *generation number*.

C. Slice & Splice Operation

Fig. 4 shows the steps involved in NFSlicer's Slice & Splice operations. NFSlicer uses the *payload* table to store the sliced payloads, and a corresponding *metadata* table to keep track of occupancy and ensure correctness while splicing. Each table has N entries and an index points to the next available entry. N is provisioned to comfortably accommodate the bandwidth-delay product of maximum arrival rate of *slice-worthy* packets and the maximum expected average service time of the target deployment's shallow NFs. *Slice-worthy* denotes the size below which slicing yields diminishing performance gains. For a target 100G system, we experimentally found this value to be 500B, which we denote as *threshold* (THR).

Table entries are allocated in FIFO order—i.e., the index is incremented after each arrival of a packet of size $\geq$ THR. Each entry in the metadata table comprises a generation number and a Time-To-Live (TTL) field. TTL enables a low-cost stale-entry garbage collection mechanism; as soon as a stored payload's TTL hits zero, the entry is discarded. The generation number is used to verify—on a packet's egress—that the payload retrieved from the payload table is a correct match.

a) Slice operation: The thick lines in Fig. 4's top half show the workflow in NFSlicer's slice operation for the common case. When a packet arrives on the ingress path, NFSlicer first checks if the packet is large enough to justify slicing (size $\geq$ THR). If so, NFSlicer creates a token containing the current index of the entry where the payload will be placed, and increments this index to point to the next empty entry. It then splits the packet into its corresponding header and payload, and stores the payload and DSCP field in the *payload* table. Finally, NFSlicer marks that the packet has been sliced by setting the packet header's DSCP field to $0b111111$ and extends the header with the token before sending it to the server for NF Processing. A packet may be forwarded to the server without getting sliced for two reasons: (i) it is not slice-worthy (i.e., size $<$ THR); or (ii) the current index does not point to an available entry—i.e., not finding readily available space in NFSlicer's structures does *not* cause a packet drop.

b) Splice operation: The thick lines in Fig. 4's bottom half show the workflow in NFSlicer's splice operation for the common case. Once the packet arrives from the server for transmission, NFSlicer checks the DSCP field to confirm if the packet was previously sliced. A non-sliced packet is transmitted out on the wire without further steps. For sliced packets, NFSlicer removes and parses the token to obtain the index to the stored payload, restores the packet's original DSCP field and payload, transmits the reconstructed packet to the network, and resets the metadata table's entry to zero.

c) Payload timeouts: NFs may explicitly drop packets as part of their operation (e.g., block rule of a firewall). NFSlicer thus requires a garbage collection mechanism to reclaim entries of dropped packets. We use the TTL field to set a validity duration for each metadata/payload table entry. A new packet on the ingress path that finds the current index pointing to an entry with a non-zero TTL is forwarded to the server for NF processing without getting sliced. NFSlicer decrements the pointed entry's TTL field and advances the current index. As the table index operates in FIFO order and each entry's TTL is decremented on each access, the TTL represents the number of allowable table wrap-arounds before an entry is evicted. In rare events of extreme processing delay spikes on the server, an outgoing sliced packet may not find its corresponding stored in the payload table, in which case the packet is dropped. Such occurrences indicate transient system overload conditions, and dropping packets has been recently used as a mechanism for improved performance predictability and overload control for microsecond-scale latency-sensitive services [5], [51], [52]. Therefore, NFSlicer's rare TTL-induced packet drops will be masked by throttling already happening at higher levels of the network stack. Besides, latency-sensitive applications, like NFs, typically have a maximum acceptable tail latency to preserve QoS (e.g., $10\times$ of average service time). The TTL can be configured to accommodate the application's acceptable tail latency, so that any forced packet drops only occur for packets that are way overdue their acceptable response time. For example, given that the base buffering capacity is provisioned to accommodate for the expected average service time, setting TTL to 10 would allow 10 buffer wraparounds, hence allowing *at least* $10\times$ average service time residency.

d) Payload retrieval correctness: As shown in Fig. 3, the NFSlicer token consists of two fields: the 64-bit token's lowest-order $\log_2 N$ bits encode the payload index, the remaining bits encode a generation number, which is required to guarantee correct payload splicing on a packet's egress. The generation number is unique per entry and is incremented whenever a new entry is inserted into the metadata/payload table. On the egress path, the generation number encoded in the outgoing packet's token is matched against the generation number stored in the metadata tables to prevent erroneous splicing of a newer packet's payload to an older outgoing packet's header, a rare situation that arises when an entry's TTL has expired and a delayed response packet corresponding to that evicted entry is transmitted from the server.

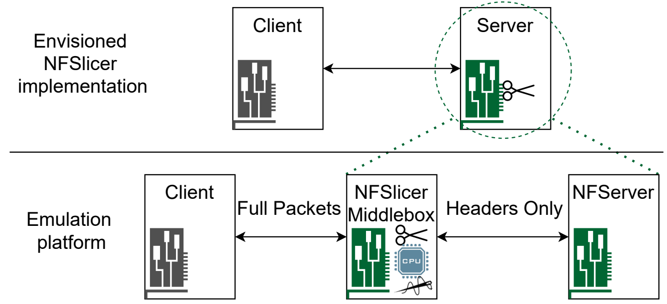


Fig. 5: NFSlicer software emulation platform developed as a DPDK application.

IV. Implementation

We envision NFSlicer as a hardware mechanism implemented on a NIC. In this work, we evaluate the Slice & Splice technique's performance effect by developing a software emulation platform, which allows us to accomplish three significant goals without undergoing the considerable engineering effort of developing a fully functional hardware prototype: (i) verify the protocol's functional correctness, (ii) evaluate the technique's performance improvement *potential*, and (iii) study the microarchitectural bottlenecks on the NF server when handling packets of different sizes. We emulate the envisioned NFSlicer functionality of the NIC in software, on a separate server. Fig. 5 shows how the emulation platform decomposes the envisioned NFSlicer-enabled server into two servers, directly connected with a 100G link: a *"middlebox"* and an *"NFServer"*. All traffic between clients and the NFServer flows through the middlebox, which performs the Slice & Splice functionality. The NFServer receives sliced instead of full-size packets of the client-originating packet flows, but executes unmodified shallow NFs.

The middlebox's Slice & Splice operations are implemented as a DPDK-based NF, which is performed in a run-to-completion fashion: a packet is received, processed and transmitted by a core before retrieving the next packet. A CPU core polls for packet arrivals on a specific port's *RX* descriptor ring, performs the Slice or Splice operation, and enqueues the transformed packet in the transmit queue. We maintain per-core arrays of `payload_t` and `metadata_t` structures (Listing 1) to store payloads and their corresponding metadata, respectively. Array sizes are specified at initialization and are provisioned to sustain the emulation platform's peak NF processing bandwidth-delay product. We further discuss array size provisioning for a hardware-based NFSlicer implementation, where buffering capacity is constrained by hardware limitations, in §VII. The middlebox performs a Slice or a Splice operation on each incoming packet, depending on its direction: client-to-server *(ingress)*, or server-to-client *(egress)*.

a) Slice operation: For each packet received on the ingress path, the middlebox determines the packet's payload size that must be sliced and stored in memory in order to forward the smallest possible packet to the NFServer. The

```c
// struct to hold the payload
struct payload_t {
  uint16_t sz; //size of payload actually stored
  uint8_t packet_dscp;  //to restore on egress
  char blk[MAX_PAYLOAD];
};

// keep track of entry occupancy
struct metadata_t {
  uint64_t generation_number;
  uint8_t ttl; //ttl = 0 indicates an empty entry
};

// table of stored payloads
struct payload_t payloads[MAX_TABLE_SIZE];

// table of metadata structure
struct metadata_t metadata_table[MAX_TABLE_SIZE];

// metadata attached to the sliced packet
struct token_t {
  uint64_t key; //index and generation number
} token;
```

Listing 1: Emulation platform's Slice & Splice structures.

middlebox maintains an index into the `payloads` array to store the payload after slicing. If the `metadata_table` is found full, the packet is forwarded to the NFServer unmodified. Otherwise, the middlebox sets the `ttl` to a predefined threshold and increments the generation number in the `metadata_table`. After storing the payload and the DSCP field in the `payloads` array, the middlebox appends a `token` to the packet containing the payload's index in the array and the generation number.

b) Splice operation: For each packet received on the egress path, the middlebox parses the DSCP field to determine if the packet has been sliced. It then parses the packet's token to obtain the index and the generation number of the corresponding payload stored in the array, which it verifies against the index and generation number in the `metadata_table`. On successful verification, the middlebox retrieves the payload from the `payloads` array; re-attaches it to the packet; restores the `packet_dscp`; clears the `ttl` field; and enqueues the packet for transmission to its destination.

c) Emulation platform performance scaling: The emulation platform parallelizes the Slice & Splice operations across multiple cores to keep up with the available network line rate. We scale its processing by employing Receive Side Scaling (RSS). We instantiate private instances of the aforementioned data structures per core to avoid synchronization overheads and employ symmetric RSS [53] to ensure that each packet is processed by the same core on its ingress and egress path.

V. Methodology

a) Experimental Platform: In order to isolate system effects as a consequence of packet sizes alone, we employ a setup as illustrated in Fig. 6. NFServer is the device under test, on which the evaluated NFs are deployed. The NFSlicer Middlebox emulates the Slice & Splice functionality, as described in §IV. To collect end-to-end latency measurements

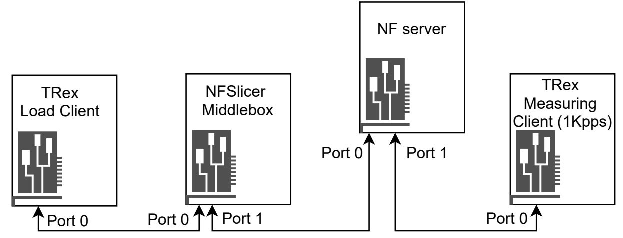


Fig. 6: Experimental setup to measure end to end latency of a client in the presence of network load.

that accurately represent server-side behavior, we employ two separate clients. The Load Client offers knobs to configure the packet size and rate, thus controlling the NFServer's operational region/utilization. The Load Client's traffic flows to the NFServer through the Middlebox.

The Measuring Client's role is to take end-to-end latency measurements that accurately reflect the server-side latency effects on our envisioned production hardware-based NFSlicer deployment, which is currently emulated in software by the Middlebox. The Measuring Client is therefore deployed with the following two provisions. First, to avoid measuring bias due to client-side queuing effects, the Measuring Client emits packets at a low fixed rate and measures each packet's end-to-end latency [58]. Second, the Measuring Client's packets do not flow through the Middlebox, to avoid biasing measurements with the high latency of the software-based Slice & Splice functionality's implementation. Despite bypassing the Middlebox, our measured end-to-end latency is representative because a hardware-based NFSlicer implementation would introduce a minuscule fixed latency overhead on each packet, as we show in §VII. Although the Measuring Client's packets do not get sliced, the vast majority of traffic received on the NFServer (>99.9% in all our experiments) originates from the Load Client and *does* get sliced. Therefore, this methodology allows us to accurately observe the systems-level effects of reduced on-server data movement, which is reflected on the resulting end-to-end latency of the Measuring Client's packets. *All latency numbers in the evaluation are reported from the Measuring Client,* averaged across three 30-second runs.

Both clients are based on the TRex scalable, open-source DPDK traffic generator [6]. Given our work's focus on large packets, the Measuring Client generates 1518B packets at 1Kpps, TRex's lowest available rate. All four server-grade machines are 2-socket Intel Xeon 4214 processors with 256GB of RAM and a dual-port 100G ConnectX-6 Dx EN Mellanox NIC, running Ubuntu 16.04.7 LTS. We disable all power management features and OS scheduling on the CPU cores involved in the experiments and only use cores and memory of the socket the NIC is directly attached to.

b) Measurement tools: In addition to TRex's in-built support for end-to-end latency measurements, we use `intel-cmt-cat` [22] for microarchitectural event measurements, and Intel's Processor Counter Monitor [2] toolset: `pcm-memory.x` for memory traffic and `pcm-pcie.x` for

PCIe utilization. We also use `DDIOTune` [11] to configure DDIO in the microarchitectural study.

c) Shallow NFs used: We evaluate four NFs which process L2–L4 headers. In order to measure end-to-end packet latency, we configure the NFs to operate in loopback—i.e., regardless of the NF processing, the ingress and egress occurs through the server's same port. Table I briefly describes each NF's functionality. The L2 forwarder, QoS metering, and Firewall, are adapted from a set of applications provided by DPDK [1]. The fourth NF implements a formally verified NAT—VigNAT [55]. In addition to the four raw NFs, we also evaluate an NF service chain comprising Firewall followed by VigNAT. We developed the NFs and NFSlicer emulation platform using DPDK version 20.02.1.

VI. Evaluation

This section first evaluates NFSlicer's achieved latency reduction for a range of shallow NFs (§VI-A). We then demonstrate the sensitivity of latency improvement to the fraction of packet payload sliced (§VI-B) and packet arrival rate (§VI-C), and compare NFSlicer to a switch-based packet slicing approach like PayloadPark [15]. Finally, we perform an extensive microarchitectural analysis to pinpoint the source of latency overhead for large packets, shedding light on the origins of NFSlicer's achieved latency improvements (§VI-D).

A. Performance Impact of Packet Slicing

Fig. 7 shows the response latency CDF for four different shallow NFs and an NF chain, ordered left to right by increasing computational intensity. All plots are at 4Mpps (Million packets per second), the maximum packet arrival rate the emulation platform can sustain before its NIC saturates, as observed by non-negligible packet drops. In all cases, the NFServer can sustain the packet rate with a single core.

We observe similar trends across NFs. First, while the minimum latency for all packet sizes is the same (∼4.5µs), the larger the packets, the higher the NF's resulting response latency. Taking L2 forwarder as an example (Fig. 7a), 512B packets exhibit a p50 response latency of 5µs, which grows to 5.4µs for 1024B packets and to 6µs for 1518B packets (see blue lines); p90 response latency grows even faster with packet size, corresponding to 5.8µs, 6.9µs, and 7.3µs for 512B, 1024B, and 1518B packets, respectively.

Second and most importantly, by slicing packet payloads and preventing large network data transfers on and off the NFServer, NFSlicer noticeably shifts the latency distribution of all packets to the left, *equalizing all of them to the best-case latency of small (64B) packets, regardless of the original packet size* (see orange lines).

NF	Description
L2 Forwarder	Modifies ethernet addresses and is used as a bridge between interfaces
QoS Metering	Measures traffic arrival rate and classifies packets into groups of corresponding rates
Firewall	Performs routing and access control
VigNAT	A formally verified Network Address Translator, which maps different IP address spaces

TABLE I: Shallow NFs used in our evaluation.

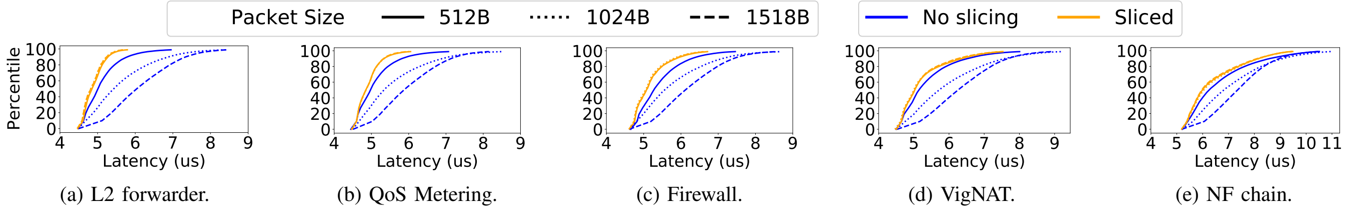
Third, the less computationally intensive the NF, the larger the latency gap between small and large packets, as processing events join data movement as a performance determinant. To illustrate, the 3.8% / 9.5% p50 / p90 latency gap between baseline and sliced handling of 512B packets in the case of L2 forwarder, shrinks to a 2.3% / 4.1% p50 / p90 latency gap for the more computationally intensive VigNAT. However, the latency gap for 1518B packets remains considerable regardless of the NF, ranging from 19.7% p50 and 18.5% p90 for VigNAT to 20.2% p50 and 28.6% p90 for L2 forwarder. NFSlicer also benefits our evaluated NF chain, which combines our two most compute-heavy NFs: Firewall and VigNAT. NFSlicer improves the 50p / 90p latency of 512B packets by 4.5% / 5.6% and of 1518B packets by 18.8% / 9.4%. *In summary, NFSlicer delivers considerable end-to-end latency reduction for all shallow NFs, with benefits growing with packet size.* Across all evaluated NFs, for 1518B packets, NFSlicer improves 50p latency by 17.0–20.2% and 90p latency by 9.4–28.6%.

B. Sensitivity to Payload Size Reduction and Comparison to Switch-Based Packet Slicing

NFSlicer is designed to reduce the amount of data moved between the NIC and the server executing shallow NFs to the bare minimum. A similar Slice & Splice approach can also be implemented in programmable switches, as demonstrated in the recent work PayloadPark [15]. PayloadPark's intended goal is to boost the goodput of switch-server links, by slicing off a fixed-size chunk (160B) of each packet's payload. A side-effect of the approach is that data movement on/off the server is also reduced, partially capturing NFSlicer's benefits. However, as we will demonstrate next, switch-based approaches face considerable limitations in terms of performance and scalability, and therefore cannot subsume NFSlicer.

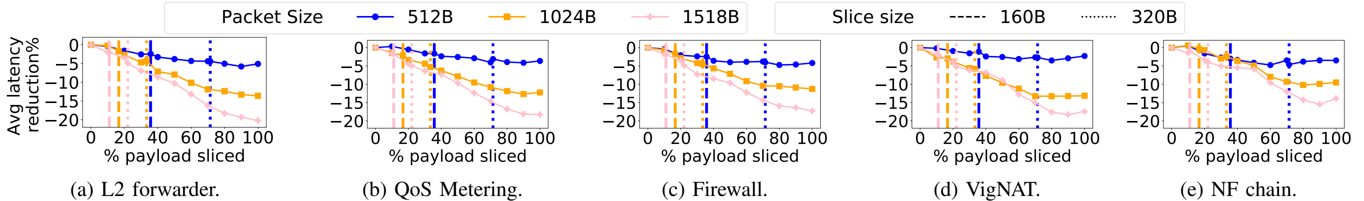
a) Performance: We modify our emulation platform to slice a configurable fraction of each packet's payload, to measure the latency impact as a function of payload size reduction. Fig. 8 shows the *average* end-to-end latency reduction as a function of the amount of payload sliced, where 0% represents the baseline case of full packet delivery (no slicing) and 100% represents NFSlicer's default operation, which slices each packet's whole payload, reducing all packets down to a fixed size of a single cache block (64B).

While the general observations regarding NFSlicer's opportunity as a function of packet size and for different NFs are similar to §VI-A, Fig. 8 additionally demonstrates that latency reduction is relative to the packet size reduction fraction—i.e., partial payload slicing will only yield a fraction of the benefits of full payload slicing. Vertical dashed lines indicate the operational point of a switch-based solution like PayloadPark, which reduces packet payloads by 160B, corresponding to 36% / 17% / 11% of 512B / 1024B / 1518B packets, respectively. While slicing 160B captures most of the benefits for 512B packets, it misses out on significant opportunity for larger packets. For



(a) L2 forwarder. (b) QoS Metering. (c) Firewall. (d) VigNAT. (e) NF chain.

Fig. 7: Latency CDFs for packets of different sizes processed by a range of shallow NFs at a fixed arrival rate of 4Mpps: baseline (i.e., No slicing) versus sliced with NFSlicer. The NF chain comprises Firewall followed by VigNAT.



(a) L2 forwarder. (b) QoS Metering. (c) Firewall. (d) VigNAT. (e) NF chain.

Fig. 8: Average latency reduction compared to no slicing as a function of fraction of payload sliced for shallow NFs at 4Mpps. Vertical lines mark the slicing fraction corresponding to 160B (PayloadPark [15]) and 320B of sliced payload per packet.

instance, for the L2 forwarder NF (Fig. 8a), slicing 11% of a 1518B payload only reduces latency by 2%, instead of 17% with NFSlicer's full payload slice.

PayloadPark's 160B slicing capability is partially implementation-specific and not a fundamental upper bound for every possible switch-based implementation. However, switches have rigid performance constraints and the maximum amount of data they can slice without sacrificing throughput will always be bound by their SRAM interface's width and their provisioned RMT pipeline's number of stages. To illustrate, a hypothetical switch-based implementation with double the payload slicing capability (320B instead of 160B, marked by vertical dotted lines in Fig. 8) roughly doubles the latency improvement potential of switch-based solutions, but still leaves NFSlicer with a considerable headway. Using the same L2 forwarder example, doubling slicing capability to 320B improves latency reduction for 1518B packets to 4.8%, still leaving a 3.6× gap with NFSlicer. Furthermore, while Fig. 8 shows *average* latency, the improvement opportunity is even larger for tail latency; for instance, NFSlicer's p90 latency improvement for 1518B packets is 5.6× higher compared to a hypothetical switch-based implementation slicing 320B per packet (not shown due to space constraints).

b) Scalability: In addition to the aforementioned performance limitations, switch-based approaches have inherent scalability limitations compared to NFSlicer. While a switch-based solution's hardware requirements grow with the cluster size, NFSlicer's scalability is virtually infinite, as hardware requirements on each server's NIC are unaffected by deployment scale. Fig. 9 demonstrates this fact with a first-order model, instrumented with data derived from PayloadPark [16], which offers two data points: the *average* SRAM utilization of the used 100G switch is 26% and 38% with 4 and 8 40G NIC NF servers, respectively. The most optimistic extrapolation indicates that switch-based slicing can support up to 38 40G

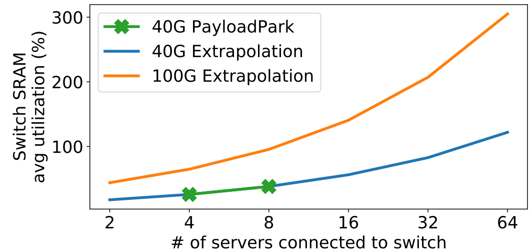


Fig. 9: Scalability study of SRAM requirements for a switch-based implementation, assuming a 100G switch. Blue/orange lines correspond to servers with 40G/100G NICs, respectively.

servers. Furthermore, a simple application of Little's Law suggests that by upgrading the NICs on the servers from 40G to 100G (to mirror our experimental evaluation setup), the switch can only support up to 8 servers.

In conclusion, although switch-based approaches like PayloadPark and NIC-based approaches like NFSlicer share mechanics, their effects on system behavior optimization only partially overlap. NFSlicer is capable of slicing a packet's whole payload—rather than only a small subset—at line rate, thus *completely ameliorating detrimental performance effects due to excess data movement*. NFSlicer also *scales* perfectly with cluster size, but, unlike PayloadPark, does not improve link goodput, as the entire packet traverses the link between the switch and the server. Given the complementary strengths of switch-based and NIC-based approaches, they can be combined to achieve all three desirable qualities: data movement minimization, link goodput improvement, and scalability.

C. Sensitivity to Packet Arrival Rate

The latency gap between small and large packet processing grows with the packet arrival rate. Unfortunately, the increased

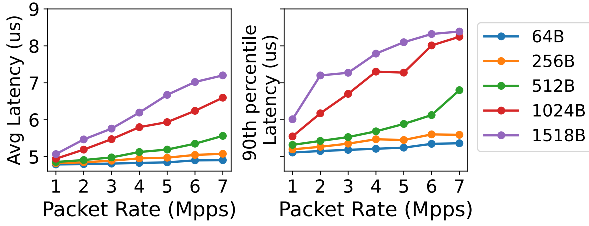


Fig. 10: Latency gap between small and large packets growing with packet arrival rate. Results for an L2 forwarder NF.

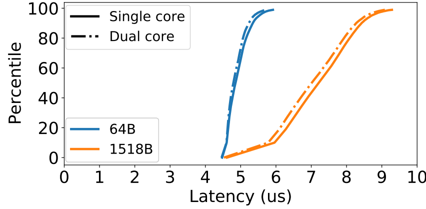


Fig. 11: L2 forwarder latency CDF for 64B and 1518B packets at 6.5Mpps. Using a dedicated core to handle the measuring client's traffic does not affect the latency gap between small and large packets.

network bandwidth requirements on our emulation platform's Middlebox (which needs to sustain 2× the target packet arrival/transmission rate) limit the peak sustainable arrival rate of 1518B packets to 4Mpps. Therefore, we demonstrate the effect of this growing gap as a function of packet arrival rate by removing the NFSlicer Middlebox and directly connecting the Load Client to the NF server, which allows us to almost double the peak packet rate to 7Mpps.

Fig. 10 shows the average and p90 latency as a function of packet rate for an L2 forwarder NF. The $1.2\times$ average latency gap between 64B and 1518B packets at 4Mpps is reduced to $1.05\times$ at 1Mpps, but grows to $1.46\times$ at 7Mpps. A similar, and more pronounced, trend appears for p90 latency: the $1.4\times$ gap between 64B and 1518B packets at 4Mpps drops to $1.08\times$ at 1Mpps, but grows to $1.55\times$ at 7Mpps.

In addition, we experimentally verify that for the packet rate range sustainable by the emulation platform (1–4Mpps), latency results of a 1518B-packet stream that reaches the NFserver after getting sliced by the middlebox are equivalent to those of a direct client-to-server 64B packet stream. In other words, from the measuring client's perspective, *loading the NF server with large packets that are sliced by NFSlicer down to 64B is equivalent to directly loading the server with 64B packets, without any prior slicing involved.* We use this equivalence throughout the next section of our evaluation to study on-server effects of full-size versus sliced large packets at a fixed packet rate of 6.5Mpps, which is significantly higher than the emulation platform's peak sustainable rate of 4Mpps.

D. Microarchitectural Study

We now embark to pinpoint the underlying sources of the performance gap between small and large packet handling.

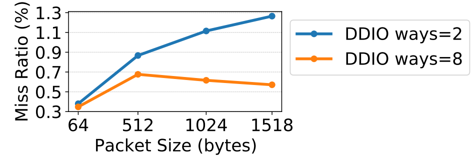


Fig. 12: LLC miss ratios for varying packet sizes.

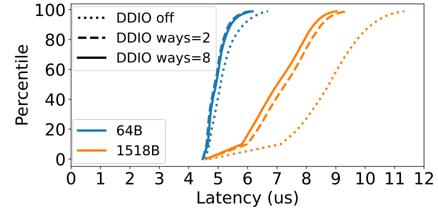


Fig. 13: L2 forwarder latency CDF of 64B and 1518B packet under different DDIO configurations.

Given the high similarity in behavior across the evaluated NFs, we focus on L2 forwarder as a representative NF for this in-depth microarchitectural study. Considering the microarchitectural components exercised by the path of packets processed by an NF, the culprit may be any of the following: the CPU, the LLC, the memory, and the NIC-processor I/O interface (i.e., PCIe). We perform a series of experiments to isolate the impact of each component.

a) CPU: Although per-packet processing requirements for our NFs of focus are insensitive to packet size, we investigate whether larger packet size introduces adverse queuing effects at the core or private caches (L1/L2), by separating the load and measuring streams to be handled on different cores (both on the socket local to the NIC). Fig. 11 shows the results. Unsurprisingly, both latency curves slightly shift to the left, as the measuring client now receives responses from a dedicated core. However, the latency improvement with the addition of a second core is minimal and, most importantly, the relative latency gap between small and large packets remains unchanged. We thus conclude that the performance gap between small and large packets *is not attributed to a processing bottleneck or contention in private caches.*

b) Cache (LLC): Modern DDIO technology [21] steers incoming packets directly into a portion of the LLC. In cases of extreme contention, incoming packets may be evicted from the LLC to memory before they are consumed by a core. This effect, known as "leaky DMA" [52] can have adverse effect on performance. We investigate if that's the case with growing packet size, by studying LLC behavior.

Fig. 12 shows the L2 forwarder NF's LLC miss ratio under two DDIO configurations: (i) the default one, which dedicates two LLC ways for network traffic injection[1], and (ii) an 8-way DDIO configuration. We observe that a 20-fold increase in packet size results in a 3× miss ratio increase under the

[1]Unless stated otherwise, this default DDIO configuration has been used throughout this paper's evaluations.

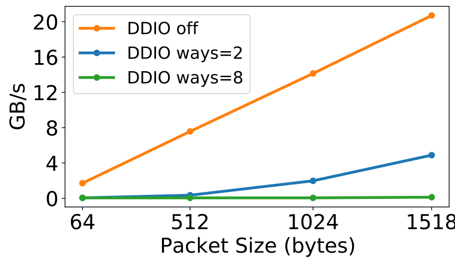


Fig. 14: Total memory bandwidth utilization under different DDIO configurations.

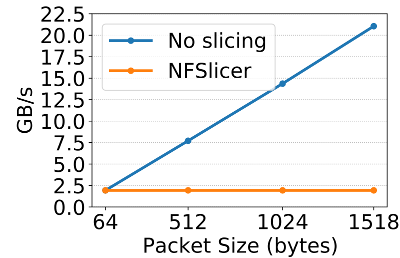


Fig. 15: NIC-generated DMA traffic over PCIe.

default 2-way DDIO configuration. However, the absolute LLC miss ratio remains negligible (less than 1%) in all cases, therefore, it cannot be a performance determinant responsible for a 55% gap in end-to-end p90 latency (see dashed lines in Fig. 13). This assessment is further confirmed by the 8-way DDIO configuration: the LLC miss ratio for large packets is 2.2× lower than 2-way DDIO (Fig. 12), but the small vs. large packet latency gap is largely insensitive to the DDIO configuration (dashed vs. solid lines in Fig. 13). We thus conclude that larger packets do not result in noteworthy LLC contention, so *LLC behavior is not the latency gap's culprit.*

c) Memory bandwidth utilization: Fig. 14 displays the memory bandwidth consumed under three different DDIO configurations: 2-way (default), 8-way, and off. The 2-way DDIO configuration offers enough cache capacity to keep all packets up to 512B LLC-resident, resulting in no memory traffic. For larger packets, some network data spills to memory, generating non-zero bandwidth consumption, which, however, is too modest to justify the latency gap between small and large packets. In contrast to 2-way DDIO, Fig. 14 shows that 8-way DDIO completely captures network data movement within the LLC, as memory bandwidth use remains zero for all packet sizes. Despite that difference, Fig. 13 shows that the 1518B-packet latency curves almost overlap for 2-way and 8-way DDIO. Hence, the latency gap between small and large packet handling *is not attributed to memory effects.*

We also study the *DDIO off* configuration as an interesting case where memory bandwidth usage *does* affect the latency gap between small and large packets. When DDIO is disabled, all network traffic moves on and off the server through memory, resulting in significant memory bandwidth usage, as shown in Fig. 14. Fig. 13 shows that putting memory accesses on the critical path has a direct impact on latency, even more so for larger packets, where the increased latency effect is amplified due to increased queuing on highly contended memory. The *additional* increase in the latency gap between small and large packets in the *DDIO off* configuration (dotted lines in Fig. 13) is attributed to memory bandwidth contention.

d) PCIe utilization: Fig. 15 shows PCIe utilization as a function of packet size. While, as expected, PCIe utilization grows linearly with packet size, this trend clearly contrasts trends of memory and cache miss ratio behavior (when DDIO is enabled). The NIC moves entire packets over PCIe to the on-server memory hierarchy, regardless of the fact that the NF executing on the CPU only operates on a small fraction of the entire packet. Aggregate data movement between the NIC and server memory hierarchy grows to 21GBps, or 67% of the theoretical peak of the PCIe3x16 interface our NIC is attached to. At such high utilization, it is common for severe queuing effects to emerge, directly hurting the latency of individual packets. In contrast, NFSlicer results in a constant low PCIe bandwidth use of only 1.9GBps, regardless of packet size.

Given our previous microarchitectural analyses showing that the processor, private caches, LLC, and memory are far from being performance bottlenecks, *we conclude that the most likely culprit for the significant latency gap between small and large packets is data movement over PCIe*. NFSlicer's Slice & Splice approach effectively alleviates this data movement bottleneck. By selectively transferring only the fraction of the packet that is needed by the deployed NF, it reduces data movement over PCIe by up to 24×, bridging the latency gap between small and large packet handling.

VII. Toward a Hardware NFSlicer Implementation

We envision NFSlicer as a hardware IP block on future NICs. Though the focus of this paper has been to present a thorough study of the system-level effects due to high network data transfer rates and the demonstration of NFSlicer's performance improvement promise, we have also embarked on preliminary hardware design and synthesis to showcase NFSlicer's feasibility for hardware implementation, and to provide area and power estimates.

NFSlicer's pipeline consists of simple logic; the main hardware cost comes in the form of payload storage, which is implemented as SRAM. In this section, we describe a hardware implementation for a 100G NIC and use Synopsys tools to acquire area, power, and timing estimates. We then outline strategies for scaling this IP block up for higher line rates.

We provision the SRAM as a static array of N M-byte entries, sized to accommodate the bandwidth-delay product of our target NF deployments. N is dictated by the maximum packet arrival rate. As NFSlicer only slices packets $\geq$ 500B, the maximum packet arrival rate at 100G is $\frac{1}{40ns}$. M is dictated by the largest packet payload NFSlicer needs to accommodate, corresponding to 1518B packets. Finally, the highest average end-to-end latency in our experiments is 9.3µs

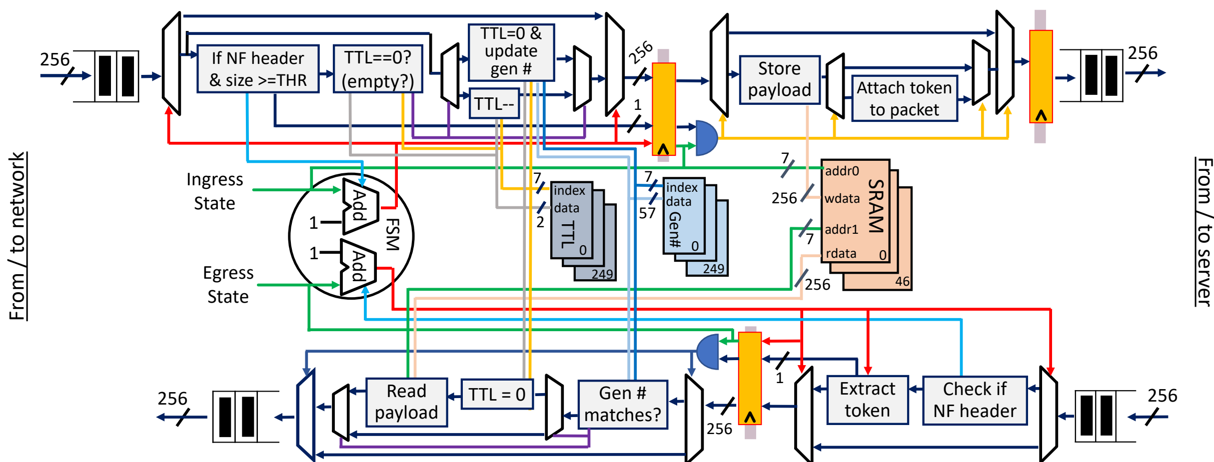


Fig. 16: Architectural diagram of ingress (upper half) and egress (lower half) Slice & Splice pipelines.

for the NF chain, which also includes the roundtrip latency between the server and the client. By over-provisioning for an average service time of 10µs, $N = \frac{10\mu s}{40ns} = 250$. Overall, our hardware design provisions $N \times M\ bytes = 355KB$ of SRAM for packet payload storage, which—to put things into perspective—is smaller than a private L2 cache of a modern multicore server.

Fig. 16 shows the high-level architecture of the ingress and egress pipelines of our NFSlicer hardware design. After verifying our design's functional correctness at the RTL level, we used the Synopsys Design Compiler and an open-source 15nm technology node library [38] to extract power, area and timing estimates from our design. We find that the NFSlicer pipeline can sustain the target line rate of 100Gbps with a 256-bit interface and a cycle time of 2.56ns (i.e., ∼400MHz frequency). The NFSlicer pipeline adds only 3 cycles on the ingress path and 2 cycles on the egress path, for a total of 12.8ns on each packet's roundtrip time. The design accounts for an area of $6.4mm^2$ and 2.4 Watts of peak power, dominated by active power.

Scaling the NFSlicer pipeline's capabilities for higher line rates (200Gbps+) is straightforward, via two design knobs: frequency and interface width. Given our design's low frequency of 400MHz, scaling it by $3-4\times$ is possible without microarchitectural changes. For a given frequency, increasing the interface width from 256 to 512 bits can double NFSlicer's sustainable line rate, without bearing significant microarchitectural restructuring of the pipeline either. For higher line rates, SRAM buffering structures need to scale to accommodate the higher bandwidth-delay product. Due to NFSlicer's logic simplicity, the pipelines' area is dominated by the provisioned SRAM and hence scales almost linearly with it. Area is typically not a major concern for modern NICs, which require large dies to drive enough pins in support of growing line rates, resulting in sparse silicon resource utilization on the die.

VIII. Discussion

New interfaces. Our microarchitectural analysis found the PCIe I/O interface as the main bottleneck for servers handling shallow NFs at high network bandwidth. The limits of PCIe under high DMA rates have been thoroughly studied before [40]. Even if new off-chip interfaces ameliorate PCIe limitations, bottlenecks due to intensive data movement may emerge in other system components. Our experiment with DDIO disabled demonstrated that increased queuing in memory can have detrimental effects, an effect that will only grow with increasing network line rates [51]. By eliminating unnecessary data movement, NFSlicer ameliorates any on-server bottlenecks associated with data deluge, regardless of the specific component they are exhibited on.

Limitations. NFSlicer stores sliced packet payloads in on-NIC memory resources. As this payload placement requires packets processed by the NFs to be transmitted by the same NIC they were received from, the technique is not directly applicable to multi-NIC servers. A possible extension could apply to dual-NIC servers that apply a known packet steering pattern, receiving packets on NIC A and transmitting them on NIC B. In such case, dedicated direct inter-NIC connectivity (e.g., Mellanox Socket Direct [42]) would allow NIC A to forward packet payloads to NIC B without utilizing on-server resources for such transfers.

Additional opportunities. Our experiments in §VI-D demonstrated that NFSlicer's benefits grow further when DDIO is disabled, because of increased latency and contention exposed by data transfers from/to memory. We expect to see an even more pronounced effect if the network ring buffers used by the NFs are mapped on a remote socket, as inter-socket links introduce considerable latency and bandwidth limitations [9].

In addition to its performance gains, we expect NFSlicer to also deliver a secondary benefit in terms of energy reduction. NFSlicer drastically reduces data movement over PCIe (Fig. 15), which should more than offset the NIC's power draw increase due to the introduction of Slice & Splice operations. Data movement in general dominates energy

consumption, and off-chip interfaces are particularly energy-hungry: transferring data over an off-chip interface consumes 1–2 orders of magnitude more energy than accessing a local memory structure (i.e., a cache) [8], [18]. An accurate energy reduction evaluation would require end-to-end measurement including a hardware NFSlicer implementation. To get a first-order estimate, we measured our entire NFServer's power draw as a function of packet size. At a 7Mpps processing rate, power draw drops from 230 to 221 Watts when the packet size is reduced from 1518B to 64B.

IX. Related Work

a) Network data movement optimization: As growing network line rates are gradually approaching data transfer rates conventionally exclusive to memory systems, on-server movement of network-injected data can drastically affect the memory hierarchy's—and, by extension, the whole system's—performance. Sutherland et al. demonstrated the negative performance impact such memory bandwidth interference can have on future systems, arguing for more sophisticated network data movement policies within the memory hierarchy [51]. A body of recent work focuses on optimizing network data placement in the last-level cache, addressing performance bumps of default DDIO behavior, such as the "leaky DMA" problem [13], [52], [54]. CacheDirector implements intelligent data placement policies to place each packet header in the LLC slice closest to the core processing it [12]. Instead of optimizing data movement within the server's memory hierarchy, NFSlicer directly decreases the volume of data moved on/off the server and within its memory hierarchy.

b) NF frameworks: The growing popularity of NF consolidation on general-purpose servers has given rise to userspace packet processing frameworks which provide ease of high-performance NF application development. NetVM [19] employs VMs to provide function isolation and shared pages for facilitating inter-VM communication. Netbricks [43] improves upon this with containers and language-enforced static checks to ensure packet isolation. Sadok et al. [46] distribute packets to cores at a packet rather than flow granularity to achieve even load distribution. Parabox [57] and NFP [50] improve NF chain processing scalability via parallel and distributed processing. NFSlicer's focus is orthogonal to these NF processing frameworks and can be combined with them to accelerate shallow NF processing of large packets.

c) Hardware offloading: Programmable (RMT) switches has seen wide applicability in offloading computation in recent years. PayloadPark [15], which we have already extensively discussed, leverages RMT switches for storing payloads. Programmable switches have also been leveraged to accelerate other network-intensive applications, such as key-value stores [24], [36]. Other prior work in this domain has improved application performance of key-value stores [30]; improved NF performance by increasing throughput and reducing latency through FPGA offloads [31]; and utilized GPUs for NF acceleration [17], [27], [56]. NFSlicer aims to improve the performance of general-purpose servers handling NFs, which have gained traction over specialized middleboxes due to their flexibility, ease of programmability, and wide accessibility.

d) Advanced and reconfigurable NICs: As general-purpose logic is running out of steam, hardware specialization is picking up, and is evidenced in growing capabilities of modern NICs, taking the form of advanced offloads or increased programmability. Hardened IP blocks implementing basic networking operations such as checksums or encryption are already commercialized in modern NICs, and there is a strong wave for offloading more advanced functionality. For instance, recent work demonstrates L5 protocol processing offloading without having to migrate the L2–L4 stack onto the NIC [44]. Dagger [29] overcomes the limitations of the PCIe interface by leveraging memory interconnects and alleviates software overheads of the RPC stack by offloading it onto an FPGA-based NIC. Ibanez et al. [20] demonstrate improvements in nano-second scale RPCs by completely bypassing the memory and cache hierarchy with a NIC-CPU co-design. NICA [10] introduces a framework that expands the SmartNIC capabilities of inline processing of application traffic to multiple tenants. A significant body of work proposes new NIC architectures [26], [34], while there have already been several successful attempts of offloading critical applications or higher-level networking functionality to advanced programmable NICs in production environments [3], [14], [45].

NFSlicer introduces the Slice & Splice operation as a basic building block and data movement optimization that can be leveraged by any shallow NF. We note that the operation could be implemented in modern programmable NICs that offer sufficient resources to sustain NFSlicer's functionality at line rate. However, dedicating programmable resources on the NIC to implement NFSlicer is both unnecessary and wasteful, as the required functionality involves mostly storage and very simple logic, making it a great fit for a hardened IP block.

Finally, for systems featuring programmable NICs, the Slice & Splice operation represents a new distinct component that an NF deployment framework [25] may determine to offload to the NIC instead of entire NFs, as it is often infeasible to fully offload all NFs in multi-tenant environments [32]. We believe that implementing the Slice & Splice as a hardware-accelerated operation in the NIC is a more resource-efficient option usable by multiple NFs than offloading entire NFs.

X. Conclusion

This paper demonstrated that data movement is a first-order performance determinant for NF processing of large network packets. As shallow NFs only operate on packet headers, we identified the opportunity to directly mitigate the overhead of redundant data movement. We introduced a packet Slice & Splice operation on the NIC to reduce NIC-server data movement to only the small portion of each packet that is needed by the shallow NFs executing on the server. We developed an NFSlicer emulation platform and showed an improvement on the median and 90th percentile latency by 17–20% and 9–29%, respectively, for a range of shallow NFs. We further showed that for higher packet rates

exceeding our emulation platform capabilities, the tail latency improvement potential grows to 55%. Finally, our hardware synthesis results showcased the practical feasibility of an NFSlicer implementation as a hardware IP block on next-generation NICs.

REFERENCES

[1] DPDK Sample Applications. https://doc.dpdk.org/guides/sample_app_ug/index.html.

[2] Processor counter monitor (pcm). https://github.com/opcm/pcm.

[3] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 7:1–7:13, 2016.

[4] Center for Applied Internet Data Analysis (CAIDA). The CAIDA UCSD Anonymized Internet Traces - equinix-nyc 20190117-130000. https://www.caida.org/catalog/datasets/passive_dataset.

[5] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload Control for µs-scale RPCs with Breakwater. In *Proceedings of the 14th Symposium on Operating System Design and Implementation (OSDI)*, pages 299–314, 2020.

[6] Cisco. TRex Realistic Traffic Generator. https://trex-tgn.cisco.com/.

[7] Cisco. Cisco Visual Networking Index (VNI) Complete Forecast Update, 2018. https://www.cisco.com/c/dam/m/en_us/network-intelligence/service-provider/digital-transformation/knowledge-network-webinars/pdfs/1211_BUSINESS_SERVICES_CKN_PDF.pdf.

[8] Bill Dally. Challenges for Future Computing Systems. https://www.cs.colostate.edu/~cs575dl/Sp2015/Lectures/Dally2015.pdf, 2015.

[9] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 33–48, 2013.

[10] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 345–362, 2019.

[11] Alireza Farshin. DDIOTune. https://github.com/tbarbette/fastclick/wiki/DDIOTune.

[12] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostic. Make the Most out of Last Level Cache in Intel Processors. In *Proceedings of the 2019 EuroSys Conference*, pages 8:1–8:17, 2019.

[13] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostic. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, pages 673–689, 2020.

[14] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 51–66, 2018.

[15] Swati Goswami, Nodir Kodirov, Craig Mustard, Ivan Beschastnikh, and Margo I. Seltzer. Parking packet payload with P4. In *Proceedings of the 2020 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, pages 274–281, 2020.

[16] Swati Goswami, Nodir Kodirov, Craig Mustard, Ivan Beschastnikh, and Margo I. Seltzer. Parking Packet Payload with P4. *CoRR*, abs/2006.05182, 2020.

[17] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue B. Moon. PacketShader: a GPU-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 195–206, 2010.

[18] Mark Horowitz. 1.1 Computing's energy problem (and what we can do about it). In *Proceedings of the 2014 International Solid-State Circuits Virtual Conference (ISSCC)*, pages 10–14, 2014.

[19] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 445–458, 2014.

[20] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A Nanosecond Network Stack for Datacenters. In *Proceedings of the 15th Symposium on Operating System Design and Implementation (OSDI)*, pages 239–256, 2021.

[21] Intel. DDIO - Data Direct I/O. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html.

[22] Intel. Intel Resource Director Technology. https://github.com/intel/intel-cmt-cat.

[23] Internet Assigned Numbers Authority (IANA). Differentiated Services Field Codepoints (DSCP). https://www.iana.org/assignments/dscp-registry/dscp-registry.xhtml.

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

[25] Georgios P. Katsikas, Tom Barbette, Dejan Kostic, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 171–186, 2018.

[26] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas E. Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXI)*, pages 67–81, 2016.

[27] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue B. Moon. NBA (network balancing act): a high-performance packet processing framework for heterogeneous processors. In *Proceedings of the 2015 EuroSys Conference*, pages 22:1–22:14, 2015.

[28] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan J. Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network Virtualization in Multi-tenant Datacenters. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 203–216, 2014.

[29] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)*, pages 36–51, 2021.

[30] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 137–152, 2017.

[31] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, and Peng Cheng. ClickNP: Highly flexible and High-performance Network Processing with Reconfigurable Hardware. In *Proceedings of the ACM SIGCOMM 2016 Conference*, pages 1–14, 2016.

[32] Xiaoyao Li, Xiuxiu Wang, Fangming Liu, and Hong Xu. DHL: Enabling Flexible Software Network Functions with FPGA Acceleration. In *Proceedings of the 38th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 1–11, 2018.

[33] Yong Li and Min Chen. Software-Defined Network Function Virtualization: A Survey. *IEEE Access*, 3:2542–2553, 2015.

[34] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *Proceedings of the 14th Symposium on Operating System Design and Implementation (OSDI)*, pages 243–259, 2020.

[35] Linley Group. Mellanox Accelerates BlueField SoC. *Microprocessor Report*, August 2017.

[36] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXII)*, pages 795–809, 2017.

[37] João Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Andrei Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 459–473, 2014.

[38] Mayler G. A. Martins, Jody Maick Matos, Renato P. Ribas, André Inácio Reis, Guilherme Schlinker, Lucio Rech, and Jens Michelsen. Open Cell Library in 15nm FreePDK Technology. In *Proceedings of the 2015 Symposium on International Symposium on Physical Design (ISPD)*, pages 171–178, 2015.

[39] Mellanox. Innova™ Flex 4 Lx EN Adapter Card. https://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova_Flex4_Lx_EN.pdf.

[40] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe performance for end host networking. In *Proceedings of the ACM SIGCOMM 2018 Conference*, pages 327–341, 2018.

[41] NVIDIA Mellanox. 200Gb/s ConnectX-6 Ethernet Single/Dual-Port Adapter IC. https://www.mellanox.com/products/ethernet-adapter-ic/connectx-6-en-ic.

[42] NVIDIA Mellanox. Socket Direct Adapters. https://www.nvidia.com/en-us/networking/ethernet/socket-direct/.

[43] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 203–216, 2016.

[44] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. Autonomous NIC offloads. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)*, pages 18–35, 2021.

[45] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James R. Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *Commun. ACM*, 59(11):114–122, 2016.

[46] Hugo Sadok, Miguel Elias M. Campista, and Luís Henrique M. K. Costa. A Case for Spraying Packets in Software Middleboxes. In *Proceedings of The 17th ACM Workshop on Hot Topics in Networks (HotNets-XVII)*, pages 127–133, 2018.

[47] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of The 16th ACM Workshop on Hot Topics in Networks (HotNets-XVI)*, pages 150–156, 2017.

[48] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: network processing as a cloud service. In *Proceedings of the ACM SIGCOMM 2012 Conference*, pages 13–24, 2012.

[49] Statista. Distribution of global downstream internet traffic as of october 2018, by category, 2018. https://www.statista.com/statistics/271735/internet-traffic-share-by-category-worldwide/.

[50] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. NFP: Enabling Network Function Parallelism in NFV. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 43–56, 2017.

[51] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra J. Marathe, Dionisios N. Pnevmatikatos, and Alexandros Daglis. The NEBULA RPC-Optimized Architecture. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, pages 199–212, 2020.

[52] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina J. Argyraki, Sylvia Ratnasamy, and Scott Shenker. ResQ: Enabling SLOs in Network Function Virtualization. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 283–297, 2018.

[53] Shinae Woo and KyoungSoo Park. Scalable TCP Session Monitoring with Symmetric Receive-side Scaling. *Technical report KAIST*, 2012.

[54] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. Don't Forget the I/O When Allocating Your LLC. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*, pages 112–125, 2021.

[55] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina J. Argyraki, and George Candea. A Formally Verified NAT. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 141–154, 2017.

[56] Kai Zhang, Bingsheng He, Jiayu Hu, Ze ke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. G-NET: Effective GPU Sharing in NFV Systems. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 187–200, 2018.

[57] Yang Zhang, Bilal Anwer, Vijay Gopalakrishnan, Bo Han, Joshua Reich, Aman Shaikh, and Zhi-Li Zhang. ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining. In *Proceedings of the Symposium on SDN Research (SOSR)*, pages 143–149, 2017.

[58] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, pages 456–468, 2016.