Packet Order Matters!
Improving Application Performance by Deliberately Delaying Packets

Hamid Ghasemirahni[1], Tom Barbette[1], Georgios P. Katsikas[1],
Alireza Farshin[1], Amir Roozbeh[1,2], Massimo Girondi[1], Marco Chiesa[1],
Gerald Q. Maguire Jr.[1], and Dejan Kostić[1]

[1]KTH Royal Institute of Technology
[2]Ericsson Research

Abstract

Data centers increasingly deploy commodity servers with high-speed network interfaces to enable low-latency communication. However, achieving low latency at high data rates crucially depends on how the incoming traffic interacts with the system's caches. When packets that need to be processed in the same way are consecutive, i.e., exhibit high temporal and spatial locality, caches deliver great benefits.

In this paper, we systematically study the impact of temporal and spatial traffic locality on the performance of commodity servers equipped with high-speed network interfaces. Our results show that (*i*) the performance of a variety of widely deployed applications degrades substantially with even the slightest lack of traffic locality, and (*ii*) a traffic trace from our organization reveals poor traffic locality as networking protocols, drivers, and the underlying switching/routing fabric spread packets out in time (reducing locality). To address these issues, we built Reframer, a software solution that deliberately delays packets and reorders them to increase traffic locality. Despite introducing µs-scale delays of some packets, we show that Reframer increases the throughput of a network service chain by up to 84% and reduces the flow completion time of a web server by 11% while improving its throughput by 20%.

1 Introduction

Recent advances in networking hardware have boosted the speed of Network Interface Cards (NICs) and packet switching devices, facilitating faster Internet access [1, 2] and improving performance in datacenters [3]. At the same time, this sudden growth in networking speeds has not been followed by a similar trend in Central Processing Unit (CPU) core frequencies and memory access latencies [4, 5]. This places tremendous pressure on today's commodity server architectures. Accessing main memory for each packet is prohibitive, thus high-speed packet processing inherently requires packets *and* the instructions & data needed to process these packets to reside in cache memories to the greatest extent possible. For these reasons, recent efforts have explored ways to optimize cache utilization, for instance, (*i*) using Direct-Memory Access (DMA) or Remote DMA (RDMA) [6] to eliminating CPU involvement in the reception of incoming packets, (*ii*) with Data Direct I/O (DDIO) [7, 8] completely avoiding main memory, (*iii*) placing incoming packets into a Last Level Cache (LLC) slice as close as possible to the core responsible for handling these packets [9], and (*iv*) realizing Network Function (NF) chains *without* inter-core communication (thus eliminating LLC cache pollution) [10] and *with* whole-stack optimizations (minimizing LLC accesses) [11].

Optimal utilization of memory caches requires that packets to be processed (with a given set of instructions *and* data) arrive as close as possible in time to each other, i.e., high *temporal* and *spatial* locality of the received packet stream. In this paper, we investigate the impact of packet ordering on the performance of I/O-intensive applications. We first measure a variety of performance metrics including throughput, average processing cycles per packet, average CPU instructions per packet, etc., as functions of the level of traffic locality of a set of streams of packets . In our experiments, the relevant data is both packets belonging to *the same flow* and the metadata that is associated with them. Our investigation reveals an unexpected sharp performance degradation (up to a factor of 3×) with even the slightest lack of temporal and spatial traffic locality for packets that could have been processed using the same instructions and data. As an example, we discovered that the number of CPU cycles per packet for an `iperf` server were reduced by a factor of $2-3\times$ when packets arrive in small bursts of 5 packets belonging to the same flow as opposed to bursts of a single packet.

In practice, there are several hindrances to cache-optimized I/O processing. First, slow NICs at the client do not produce bursts of packets that will arrive "back-to-back" at a receiver with a faster NIC. Moreover, the multiplexing of different traffic flows along the path from a client to a server results in packets belonging to a client's flow being spaced apart (i.e., interleaved with other flows), thus diminishing locality. Even worse, we observe the existence of an increasing *friction* between emerging networking trends, which advocate that congestion control mechanisms pace packets, i.e., spread packets in a flow apart from each other as much as possible to minimize the risk of congestion in the network (see §3), and the desire to process incoming packets in memory caches to the greatest extent possible (due to trends in computer architecture) [12]. To understand whether real-world traffic exhibits sufficient ordering, we analyzed a real-world traffic trace from one of the packet gateway interfaces of our organization. This traffic exhibits a very low level of spatial locality, as more than ~60% of the packets belonging to the same flow are interleaved with packets belonging to other flows, which is far from ideal conditions for cache-optimized packet processing.

These apparently completely opposite requirements of (i) pacing traffic for better network-level statistical multiplexing and (ii) processing packets in bursts for better cache effectiveness calls for a solution that satisfies both requirements at the same time. Based on the above, we explore the counter-intuitive idea of *increasing packet processing throughput by deliberately delaying and reordering packets before they reach the application running on the server(s)*, thus rebuilding high traffic locality. We built Reframer, a network function that leverages this idea, to buffer and reorder packets between different flows. By introducing Reframer at the *destination* network (or directly at an end server), we (i) maximize the number of subsequent cache hits in the servers, thus reducing the processing time for each burst and (ii) are compatible with the emerging pacing-based congestion control mechanisms (e.g., BBR [13]) as we do not affect the pacing of the packets across the Internet. Reframer can be deployed on the same server where one needs to increase cache hit performance (e.g., CPU core and/or SmartNIC) or upfront as part of a network function service chain to improve the throughput of the service chain itself by up to 60% (see §5.2) and subsequent web servers throughput by 20% while reducing the flow completion time by 11%, despite delaying the individual packets. Moreover, we show that Reframer improve performance an order of magnitude more than flow-oblivious batching [14], showing the need to increase *per-flow* spatial locality.

Contributions. In this paper, we:

- Unveil that trends in networking, spreading packets apart, are antithetical to today's high-performance computer architectures, which require bursty communication to efficiently use cache memories for high-speed networking.
- Systematically measured the performance degradation due to the lack of spatial locality in the streams of packets processed by servers for a variety of I/O-intensive applications (including large data transfers and network functions). Our results show significant performance degradation, up to a factor of $2-3\times$, mainly due to cache misses (§2).
- Analyzed the levels of spatial and temporal locality in real-world traffic captured between our organization and our ISP. This traffic shows poor locality, which leads to sub-optimal performance at each of the servers (§3).
- Built a Reframer prototype to reorder packets, thus exploit servers' caches when processing packets at high speed (§4). Reframer improves the throughput and latency of chained NFs by up to 84% and 46% respectively, using a realistic packet trace and various Reframer deployments (§5).

2 How Much Does Order Matter?

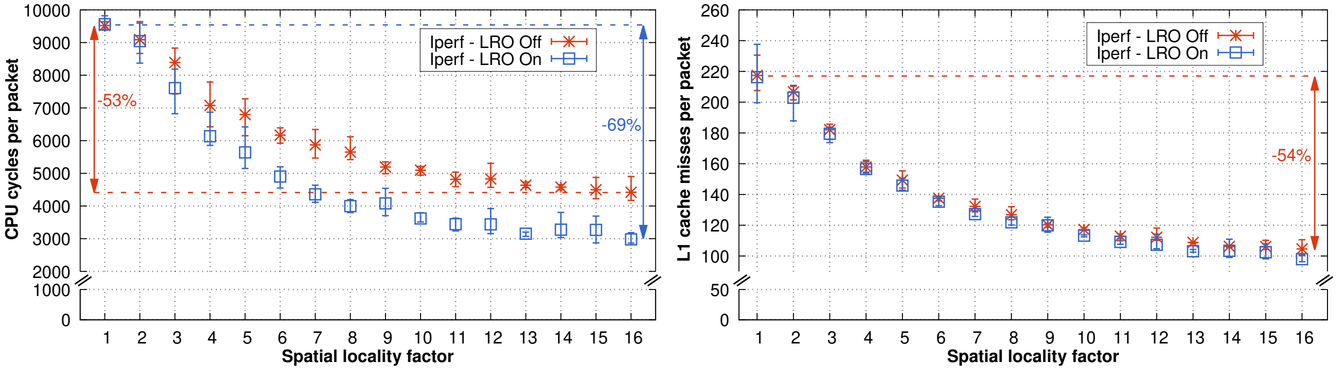
This section shows how explicit packet ordering increases temporal and spatial locality and, consequently, boosts the performance of real-world applications. Our results show that, when packets belonging to the same flow are interleaved by even a few other packets, the latency of a packet processing application may increase by more than 2× because of a higher number of cache misses and executed CPU instructions. These results motivate our Reframer system, whose goal is to build *per-flow* batches of packets that can be submitted to the servers, as opposed to batches of *arbitrary* packets belonging to *different flows* as in state-of-the-art software switches (e.g., Batchy [14]).

The experimental methodology used in this section is described in §2.1. We decompose the effects of packet ordering into three categories: network stack effects (§2.2), software switching effects(§2.3), and more advanced NF effects (§2.4).

2.1 Experimental Setup

Testbed. All the experiments in this section use the same testbed. Two back-to-back interconnected servers, each with a single-socket 8-core Intel® Xeon® Gold 5217 (Cascade Lake) CPU clocked at 2.3 GHz and 48 GB of DDR4 RAM at 2666 MHz. Each core has 2×32 KiB L1 (instruction & data caches) and 1 MiB L2 caches, while one 11 MiB LLC is shared among the cores. Each server has a dual port 100 GbE Mellanox ConnectX-5 NIC with firmware version 16.28.1002. Hyper-threading is enabled on both servers and the Operating System (OS) is the Ubuntu 18.04 distribution with Linux kernel v5.3. One server acts as a traffic generator and receiver while the other server is the Device Under Test (DUT). We also utilized the Linux `perf` tool on the DUT during the execution of the experiments to monitor CPU performance counters (e.g., CPU cache misses).

Spatial locality factor (*SLF*). We define *SLF* as the average number of packets, in the same flow, that arrive back-to-back at the DUT. For example, if there are three flows (A, B, and C) and $SLF = 1$, the DUT receives packets in the pattern "ABCABC...". For $SLF = 2$, the pattern is "AABBCC...".



(a) CPU cycles per packet.

(b) L1 cache misses per packet.

Figure 1: Impact of packet spatial locality on the performance of an `iperf` server, with and without LRO.

2.2 Network Stack Effects

Packet ordering has a profound impact on the performance of general purpose network stacks and their applications, especially TCP receive-side processing. In these experiments, we show that lack of traffic locality greatly degrades CPU utilization (up to a factor of 3×) even when sophisticated TCP accelerations are used.

In these experiments, we use Linux `iperf` [15] to establish 128 TCP connections (with 1500 B packets) to the DUT that runs an `iperf` server. The duration of each test is 15 seconds.

We utilize the Linux traffic control mechanism (`tc`) on the client side to synthetically order the sending packets with a given value of *SLF*. We restrict the sending rate to ~8 Gbps, as forcing a real TCP stack to exhibit a specific *SLF* at high speeds is extremely hard. On the DUT side, we restrict `iperf` to use only one core to clearly delimit the benefits of packet ordering from potential artefacts introduced by parallelism.

Lack of locality makes TCP accelerations ineffective. A variety of TCP accelerations have been devised to mitigate the effects of the increasingly faster NICs' transmission speeds on the relatively stable CPU speed. In this experiment, we show that the most notable of these accelerations, i.e., Large Receive Offload (LRO), is ineffective with low traffic locality.

Ideally, LRO should combine *SLF* consecutive packets of the same flow received at the NIC into a single "super-frame", removing all the Ethernet & IP headers from the merged packets and possibly coalescing redundant packets, such as TCP acknowledgements. However, interleaved packets from different flows prevent LRO from merging consecutive packets which leads to inefficiency of LRO.

The blue boxes of Fig. 1a show that LRO performance is improved significantly when the spatial locality factor increases from 1 to 16, i.e., more consecutive packets in a flow arrive at the DUT. This increase in *SLF* reduces the number of CPU cycles per packet by 69% (from ~10k to ~3k), which shows low traffic locality harms TCP acceleration by LRO. Even without LRO (red boxes in Fig. 1a), the number of CPU cycles per packet decreases by 53% with an increasing *SLF*.

Two explanations for the benefits of spatial locality are:

① **Fewer cache misses.** Ordered packets increase L1 cache hit ratio as common per-flow data structures are fetched only once for all packets. Fig. 1b shows that the number of L1 cache misses per packet decreases by 54% when packets are processed back-to-back. Particularly, we observed an increase in performance for the "__inet_lookup_established" Linux kernel routine. This function performs a lookup in the listening sockets hash table to assign the received packet to the corresponding socket. The improvement is identical regardless of whether LRO is enabled or not and simply depends on having a better packet locality.

② **Fewer CPU instructions per packet.** Since `iperf` uses multiple threads to serve clients' requests, when *SLF* is small, the scheduling routines of the Linux kernel are called more frequently to switch among `iperf` threads. By increasing *SLF*, each thread is able to handle multiple consecutive packets (ideally *SLF* packets) within a single scheduling round of the Linux kernel. Consequently, the number of flow handling routines and executed CPU instructions decreases dramatically with or without LRO enabled (see Fig. 2). LRO further reduces the average number of CPU instructions per packet thanks to the creation of super-frames of packets.

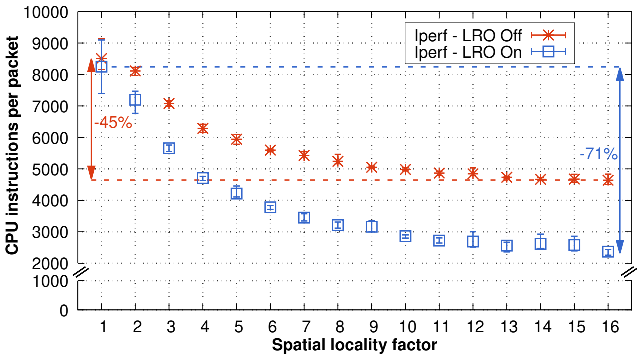
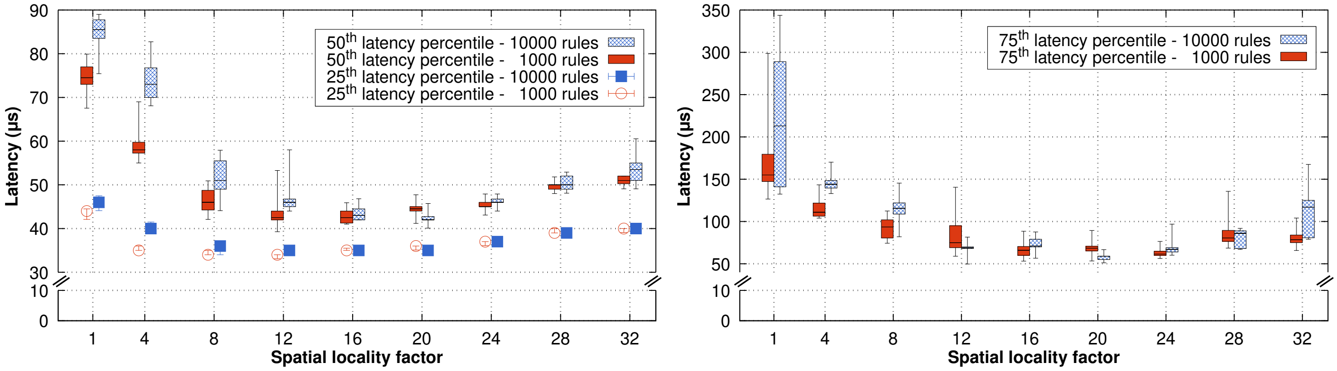


Figure 2: Impact of packet spatial locality on CPU instructions per packet of an `iperf` server, with or without LRO.



(a) 25th (errorbars) and 50th (boxplots) percentile latencies.

(b) 75th percentile latency.

Figure 3: Impact of spatial locality on the forwarding performance of OVS 2.13.9 using the Linux kernel v5.3 data path.

Takeaway. From this experiment we conclude that the performance of today's high-speed networking applications is highly dependant on the spatial locality of the received packets, as this impacts cache-miss ratios and the number of CPU instructions per packet. Based on Fig. 1a, we observe that systems without LRO acceleration but with good spatial locality of packets (i.e., $SLF = 16$) perform *better* than systems with LRO but with poor locality of packets (i.e., $SLF < 5$), making it beneficial to process ordered streams of packets.

2.3 Software Switching Effects

This section quantifies the effects of locality on the performance of the kernel-based Open vSwitch (OVS) [16]; a widely deployed production quality multi-layer software switch. Many Virtual Machine (VM) and container-based cloud platforms (e.g., VMware NSX-T [17], OpenStack [18], Red Hat's OpenShift [19], and Kubernetes [20]) use OVS.

OVS classification pipeline. Upon a packet's arrival, OVS employs a multi-stage classification pipeline. The first stage is a 2^{13} entry Exact Match Cache (EMC) for frequently used flows. This cache uses a 32-bit hash of the packet's header, which can be the Receive-Side Scaling (RSS) hash, as a key mapped to a rule for the corresponding packet. In OVS 2.10, a second classification stage called Signature Match Cache (SMC) was introduced as an experimental feature. This cache stores a 16-bit signature for each flow along with a corresponding 16-bit index into a flow table (with up to 2^{16} rules), a total of 32 bits; hence, it is more memory efficient than EMC, which stores the entire forwarding rule.

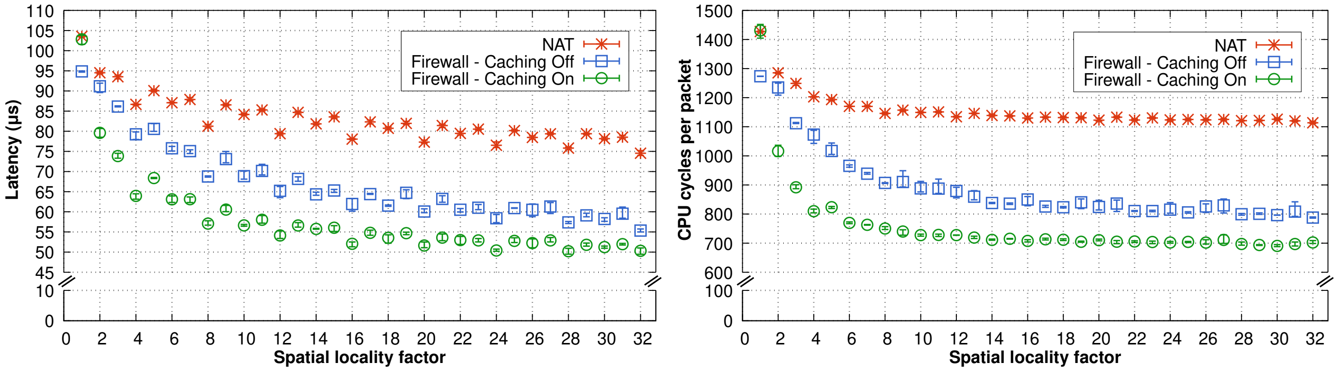
If neither of the first two cache levels matches an incoming packet, then that packet is classified by the kernel's Megaflow cache [21]. This cache is based on the Tuple-Space Search (TSS) algorithm [22] that uses more aggressive bitwise wildcarding to aggregate multiple flows into a single match. Finally, a miss in the Megaflow cache results in a packet redirection to the "slow path", where packets traverse a pipeline of OpenFlow tables to derive their corresponding actions.

OVS setup choice. Due to the fact that the EMC is an n-way associative cache (similar to a modern CPU cache), only n out of 2^{13} entries can be used to store any given flow. In OVS version 2.13.9 $n = 2$, implying this cache will likely exhibit high contention even when the number of flows is much smaller than the EMC. Measurements of these OVS caching schemes showed that EMC does not yield the expected levels of performance improvements over the SMC [23]. Specifically, EMC slightly outperforms the SMC *only* with low numbers of flows (< 200), while SMC offers higher performance with more flows [23]. We verified this through our own experiments, hence we disable EMC to achieve higher performance.

OVS experiment. We deployed OVS 2.13.9 on the DUT with a data path through the Linux kernel v5.3 of the DUT. The forwarding behavior is defined by two sets of OpenFlow v1.4 rules with 1k and 10k entries. These rules classify input packets based on their source and destination Ethernet and IP addresses and forward matching packets toward the traffic receiver through the same port (i.e., the Mellanox port of the DUT attached to OVS). Only one rule in each rule set matches the input traffic. We used a Data Plane Development Kit (DPDK)-based traffic generator to inject a trace of 10k User Datagram Protocol (UDP) flows, where each flow consists of 1500-B packets, at the rate of 5.5 Mpps $\approx$ 66 Gbps*. Fig. 3 shows the performance of the kernel-based OVS classifier, focusing on the 25th & 50th (Fig. 3a) and 75th (Fig. 3b) latency percentiles.

Packet ordering greatly benefits OVS's caching scheme. When no particular locality is enforced (i.e., $SLF = 1$), the 75th latency percentile (see Fig. 3b) ranges between 132 µs-343 µs and 126 µs-300 µs for 10k and 1k rules, respectively; while lower latency variance is observed in Fig. 3a for the 25th and 50th latency percentiles. However, both latency and its variance substantially decrease with increasing SLF for both rule sets. The greatest improvement is observed for $SLF \in [20, 24]$, where packet locality results in $2.5 - 5\times$

*Similar results occur for TCP packets. With 64-B packets, the effect of packet ordering is less profound, but still relevant.



(a) End-to-end latency.

(b) CPU cycles per packet.

Figure 4: Impact of traffic spatial locality on the packet processing latency and the CPU cycles per packet performance of a NAT and a firewall (with and without rule caching) NFs.

lower 75th latency percentiles, 2× lower medians, and 15-22% lower 25th latency percentiles with negligible latency variance. For higher values of the spatial locality factor (i.e., $SLF \in [28,32]$), we observe a slight latency increase compared to the lowest attainable latencies shown in this figure. This behavior is not observed in the other experiments in this section, suggesting the limits of packet ordering be studied on a case-by-case basis.

10k rules at the cost of 1k rules. An equally important benefit of this use case is shown in the case of $SLF \in [20,24]$, where the red and blue boxplots and error bars in Fig. 3 exhibit very similar ranges. This means that packet ordering amortizes the additional cost of a 10x larger classifier (i.e., 10k vs. 1k rules) by making the most out of OVS's caches.

2.4 Network Functions' Effects

In addition to network stacks (§2.2), packet locality may also affect more advanced NFs. To investigate this, we implemented two NFs in FastClick [24], a stateless firewall and a (stateful) Network Address Translation (NAT)[*]. Unlike §2.2, we allocate two cores per NF with one RX queue per core to show that the benefits of packet locality is not limited to single-core scenarios. We will further discuss the impact of number of RX queues on the DUT performance in §5.1. In these experiments, the traffic generator emulates 10k clients sending a total of 20 million 1-KB UDP packets to the DUT with a total rate of ~50 Gbps (6.2 Mpps) and a given spatial locality factor SLF. Fig. 4 shows the average end-to-end latency and the number of CPU cycles per packet for these two applications.

NAT NF case. We deployed the NAT NF on the DUT. Fig. 4 shows that the end-to-end latency decreases from 103 µs to 74 µs as the spatial locality factor increases from $SLF = 1$ to $SLF = 32$. When $SLF = 1$, some packets are dropped since for each packet, the CPU must wait for the many cycles it takes to fetch the appropriate NAT table's row from the memory, greatly decreasing the available useful processing time and the capacity of the NF to serve incoming packets. In contrast, when input packets are partially ordered by flow, the NF amortizes the cost of this NAT table lookup over several consecutive packets within the same flow, thus reducing the average processing time needed to serve each packet.

Firewall NF case (without software-based rule caching). We deployed a firewall NF implementing a tree-based Access Control List (ACL) with 20k rules on the DUT. We consider two different variants of this firewall. The first variant assumes *no* rule caching, thus it executes the matching algorithm for each incoming packet. Since all packets of the same flow typically match the same rule, then with an increasing spatial locality factor, we expect a reduction in the frequency of fetching data (rules) from main memory into the system's cache(s). The blue boxes in Fig. 4 show similar trends as in the previous experiment, i.e., an increasing spatial locality factor improves the performance of the firewall in terms of both average end-to-end latency (Fig. 4a) and number of CPU cycles per packet (Fig. 4b).

Firewall NF case (with software-based rule caching). The second variant of this firewall NF implements a simple in-memory rule cache. This cache stores the hash of the last served packet and the matched rule. For each incoming packet, the firewall calculates the packet's hash value, and if it is the same as the entry in the cache, then it assumes that the packet will match the same rule as the previous packet. However, if after executing the rule the new packet does not match the rule, then the cache will be updated with a new matching rule and a new packet hash. The green circles in Fig. 4 show faster convergence to the minimum values compared to the firewall *without* caching as the firewall's cache matches an increasingly large fraction of input packets (i.e., $SLF - 1$ packets for a given SLF) without invoking the firewall's classifier.

Packet spatial locality analysis. Looking closely at the per-packet CPU cycle curves shown in Fig. 4b, we note that the

[*] We also deployed a chain of NFs on the DUT as a complementary experiment in Appendix A.1

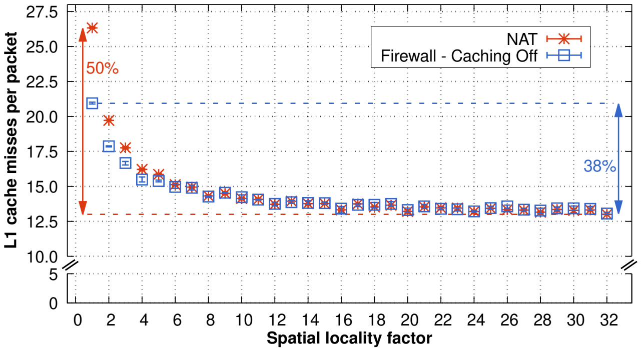


Figure 5: Impact of spatial locality on the number of L1 misses per packet for a Firewall (w/o caching) and NAT NF.

data fits an equation of the form $cost = \alpha * (1/SLF) + \beta$, where β is the CPU cost of processing the data that has already been accessed and is in the cache, hence it is the asymptotic limit when SLF is large. In contrast, $\alpha * (1/SLF)$ is a weighted version of the cost of getting the data that can be shared, e.g., when $SLF = 2$, the cost per packet is amortized over two packets.

In the case of the NAT, when $SLF > 1$ the main cost is the lookup of the appropriate replacement values in the NAT table and this lookup only has to be done once for the first packet, hence $\alpha \approx 1$ times the cost of this lookup. In the case of the firewall, we expect that for a given number of firewall rules F, $\beta \propto \gamma * F$ when the firewall rules cannot be cached (i.e., when the rules cannot fit into the cache), hence the firewall rules have to be repeatedly loaded and hence the cost cannot be shared (i.e., $\alpha \approx 0$). However, we see that this is not the case in Fig. 4, as an application still benefits from processor-based caching of the data even *without* software-based rule caching.

Serving packets at the speed of L1 cache. We now highlight the fundamental role played by core-specific L1 cache in enhancing the performance of the above NFs. To measure cache-related events, we utilized the Linux `perf` tool during the execution of the experiments shown in §2.4. Since the NFs' data size (NAT table and firewall rules) are smaller than the LLC and L2 capacity, we see almost no LLC and L2 misses; hence, the reduction in the number of CPU cycles is mostly due to better utilization of the L1 cache.

Fig. 5 shows the effect of locality on the number of L1 cache misses for both the NAT and firewall experiments. In both cases, we observe a substantial decrease in the number of L1 cache misses. Our analysis reveals that we can observe the effects of ordering even on the L2 and LLC misses by deploying a memory-intensive NF (e.g., Deep Packet Inspection (DPI)) or a chain of multiple NFs on the DUT (Appendix A.1). Our results demonstrate that better utilization of core-specific caches is the key for increasing the NFs' performance *and* ordering packets minimizes cache misses.

2.5 Summary

In this section, we explored the effects of spatial locality of network data by conducting experiments across Linux network stack and DPDK-based stateless & stateful NFs at various levels of a system's software stack. The common denominator of this study is that packet ordering greatly increases the utilization of a server's memory hierarchy (mostly CPU caches), which in turn results a substantial improvement in key performance indicators, such as latency, throughput, and CPU utilization.

We leverage these insights to design a system that vertically (i.e., hardware to application layer) exploits the benefits of packet ordering (see §4) and demonstrate complementary results using additional real world applications (see §5). Before this, we investigate whether today's Internet traffic exhibits a low or high spatial locality factor (see §3).

3 Packets Order in Real-world Traffic

This section analyzes a trace from our organization (i.e., a university) to understand the spatial & temporal locality in realistic traffic (§3.1) and explores opportunities to increase traffic locality by reordering packets (§3.2). Our analysis shows that >60% of the packets belonging to a flow are interleaved with packets of other flows, hence non-ideal for high-speed packet processing (based on §2). Moreover, today's networking trends further exacerbate this – as novel congestion control mechanisms (e.g., BBR [25], Timely [26], HULL [27], and Carousel [28]) advocate pacing packets to fight "bufferbloat", i.e., keeping queue occupancy in routers' buffers as low as possible. Even the built-in self-clocking of traditional TCP congestion control mechanisms [29], which inherently spreads packets out over time to avoid congesting a link, is harmful to cache-optimized high-speed network communication. In §4, we advocate rebuilding per-flow traffic bursts as close as possible to the servers that process them.

Trace statistics. We captured 28 min of traffic from our campus to & from our upstream network provider. The outgoing traffic (i.e., from the campus toward the Internet) had 420 million packets with an average size of 1069.43 B and the incoming traffic (i.e., from the Internet toward the campus) had 378 million packets with an average size of 882.82 B. Fig. 6 shows the TCP flow size distribution for this traffic. In the rest of this document, we refer to the outgoing and incoming traffic as the TX and RX traces, respectively.

3.1 Spatial & Temporal Distance

The performance benefits of packet spatial locality were shown in §2 with the greater the number of consecutive packets belonging to the same flow (i.e., the spatial locality factor), the greater the benefits. Additionally, we concluded that even a small spatial locality factor (e.g., $SLF = 5$) could yield a

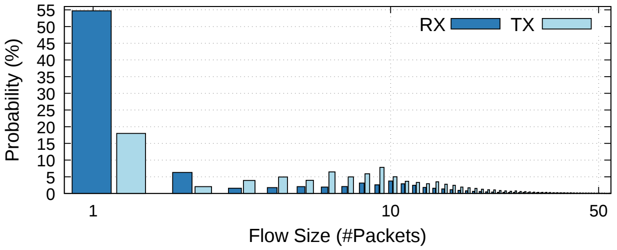
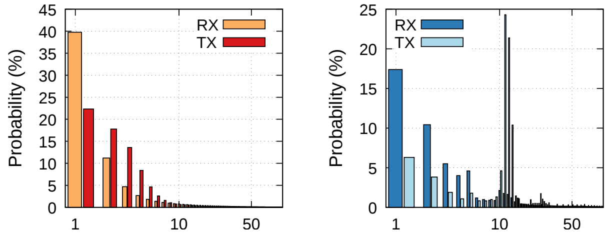


Figure 6: TCP flow size distr. of the analyzed trace with a log. x-axis. The RX trace has ∼ 4M flows; the min., avg., and max. flow sizes (in #packets) are 1, 63, and ∼ 29M, resp. The TX trace is composed of ∼ 2M flows; the min., avg., and max. flow sizes (in #packets) are 1, 137, and ∼ 68M, resp.



(a) Spatial distance (#Packets).

(b) Temporal distance (µs).

Figure 7: Distribution of the spatial & temporal distance for the campus trace. (Note that the x-axis is logarithmic).

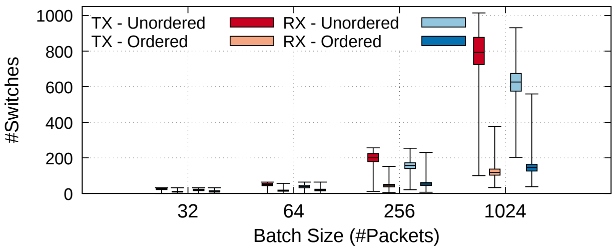


Figure 8: Number of per-flow switches for different batch sizes (selected according to [14]: 32/64 for Linux kernel and DPDK; 256 for VPP; and 1024 for GPU/NIC offload).

significant improvement, as was shown in Fig. 4. However, the improvements depend on the traffic's actual spatial locality. Therefore, in this section, we examine how *unordered* the trace from our organization is. To do so, we calculate the spatial and temporal distance of packets in every TCP flow. *Spatial distance* shows the number of packets between two consecutive packets of the same flow and can be used to assess opportunities to exploit cache memories. The higher the spatial locality, the greater the number of opportunities to increase cache-hit ratios. *Temporal distance* measures the time between two consecutive packets of the same flow and can be used to estimate how long one would have to wait for another packet in order to reorder packets and increase spatial locality. Fig. 7 shows the histogram of these metrics for the campus trace. These results do not consider single-packet flows[*], as these metrics are undefined for such flows.

Spatial distance. Fig. 7a shows that the spatial distance of the per-flow packets are larger than one packet in ~60% of the RX trace (without single-packet flows) and ~75% of the TX trace (without single-packet flows) – i.e., there is *at least* one packet between consecutive packets of the same flow. The rate of our campus trace is ∼2.2 Gbps, which underestimates the values reported for the spatial distance. In networks with higher rates (e.g., multi-tens- & multi-hundred-gigabit rates), the spatial distance would *intuitively* be much larger, which further reduces the locality. As shown in §2, this lack of spatial locality can dramatically degrade performance, up to factor of 3×. Fig. 8 shows the number of switches across different flows that an application should theoretically perform when processing different batch sizes of packets. The number of switches can be more than 5× larger when the packets are unordered. Frequent switching could cause detrimental performance events (e.g., context switches and/or cache evictions), the number of which depends on the system's microarchitecture (e.g., cache hierarchy) and the application characteristics (e.g., the type of processing and the size of the per-flow state).

Temporal distance. Fig. 7b demonstrates that temporal distance between consecutive flow packets in a flow is typically smaller than a few tens of microseconds, making it possible to reduce the spatial distance by buffering packets for a short time so that they can be reordered. The potential for reordering of traffic destined/originated to/from two cloud providers is described next.

3.2 Potential of Per-flow Ordering

We identified the top hundred IP addresses of the TCP connections, which appeared in independent flows of the TX trace. From those, we select those of two popular cloud providers, referred to as $Cloud_1$ and $Cloud_2$[†]. We calculated the probability of receiving packets of the same TCP flow within different fixed-size time windows to determine whether by waiting for a short amount of time we can reorder packets to make *per-flow* batches of packets, i.e., regenerate high spatial locality. Additionally, since user-space packet processing frameworks (e.g., DPDK) use a fixed batch size for processing packets (typically 32 for a DPDK-based application), we assume that up to 32 packets per flow can be buffered[‡]. Fig. 9 shows the distribution of batch sizes for different buffering times. These results consider all flow sizes, including single-packet and mice flows which dramatically reduces the size

[*] Based upon the source addresses, we expect that some of these are likely to be part of SYN attacks.

[†] Table 1 (in Appendix A) shows the statistics of these flows.

[‡] In some cases it might be possible to buffer up to ~300 packets, see Fig. 23 (in Appendix A).

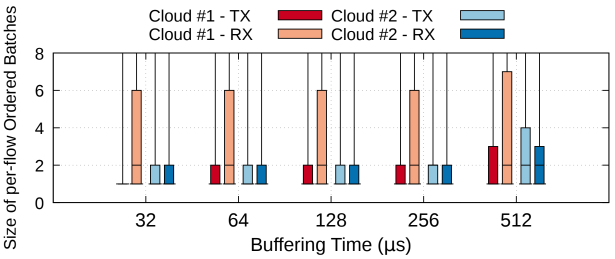


Figure 9: Impact of increasing the waiting time on the probability of receiving packets in the *same TCP flow* (i.e., packets going to the same end-host, the same core, and the same application).

of the per-flow batches. Clearly, increased buffering time is positively correlated with receiving more packets in the same flow. However, the statistical properties of traffic (e.g., a cloud service) should be taken into account when ordering packets. For instance, it is possible to make per-flow batches of size 6, even in 32-µs time frames, for 25% of the incoming traffic from $Cloud_1$; whereas 25% of the outgoing traffic toward $Cloud_2$ could only be made into per-flow batches of size 4.

Summary. This section showed that *most* of the flows in a campus trace could benefit to some extent from ordering, as it increases locality, i.e., decreases both spatial and temporal distances. However, the improvements depend on the traffic characteristics and type of service. Ordering of larger flows can potentially lead to much bigger improvements (see the tails of the box plots in Fig. 9). Therefore, a cloud provider/operator might only apply reordering to specific services and/or tune the waiting time based on the Service Level Objective (SLO) and the flow rate.

4 The Reframer Design

As we have shown in Section 2, receiving unordered packets leads to high cache misses and more CPU cycles per packet, which increases the cost of packet processing in networking devices. This section presents our proposal to achieve increased end-to-end data locality (both temporal and spatial) of packets in each flow. Our solution maximizes locality and is compatible with today's trends in Internet congestion control paradigms that pace packets. We leverage the idea of briefly buffering, delaying, and reordering the (possibly paced) incoming packets to increase spatial locality for network traffic. As a result, Reframer pays the imposed price of receiving paced packets only one time at the beginning of a NFs and applications chain instead of allowing every single NF pays that for itself. Reframer is developed as a *software* solution that uses CPU cycles to classify flows and create batches with higher locality. Following the trend of Network Functions Virtualization (NFV), advantages of software network functions like Reframer include more flexibility, faster development cycles, and nearly no resource limitation (e.g., number of per-flow queues) in comparison to hardware alternatives. On the other hand, similar to many networking software systems, the main design challenge is efficiency in terms of both time and space complexity: one needs to strike a delicate balance between the complexity of the reordering procedure, which *consumes* CPU cycles, and the gains at the application/NFs, which *saves* CPU cycles; Hence, it is crucial for Reframer to employ an optimized data structure that takes a short time and space for reordering packets regardless of incoming packets rate and the number of concurrent flows. With Reframer, the incoming packets are efficiently buffered and reordered and then delivered to their destinations. Fig.10 shows the operation of Reframer when a stream of packets belonging to three flows (i.e., green, blue, and brown) arrive at the Reframer.

Flow classification. Reframer maintains two main data structures to reorder packets: a *flow classification table* and a *flush list*. For each flow, the flow table stores the timestamp (TS) when the first packet of that flow has been added to the batch of that flow. It also stores a pointer to the list of the buffered packets for that flow. The flush list is a double-linked list that stores flow identifiers sorted by timestamp described in the flow table. Reframer updates all these data structures in constant time for a variety of operations: buffering of a packet in the flow table (when a flow entry already exists), adding/ removing flow identifiers to/from the flush list, finding the oldest flow identifier, and emitting a batch of packets. Only insertion of new flows in the flow table is not performed in constant time because of the cuckoo-hash table. Additionally, Reframer stores only a few bytes of metadata per flow that allows CPU cores to work at the speed of L1 and L2 caches.

In case the number of packets in the flush list meets a configurable limitation (*maximum burst size*), Reframer passes the batch to the scheduler.

Buffering Time. The flush list can buffer flows for a maximum amount of time, which we call the *buffering time* (T_{buff}). The optimal buffering time mostly depends on two parameters: (i) flows' average throughput and (ii) the end-to-end latency between a Reframer instance and the destination. The former parameter affects the possibility of receiving multiple packets of the same flow in a short time window. For instance, the inter-arrival time of 1000 B packets is 8 µs at 1 Gbps and Reframer can rebuild a per-flow batch with up to 8 packets by buffering packets for 64 µs. The latter parameter sets the upper bound for the buffering time., i.e., a higher end-to-end latency provides more flexibility to wait for packets. It is possible to adjust buffering time by automatically calculating both of these parameters; However, in the current version of Reframer, it should be configured manually by an operator.

Reframer collects additional information to track its efficiency, i.e., (i) it measures the amount of time that flows were being delayed without actually receiving more packets, and (ii) it records the average amount of packets per batches. These statistics could potentially enable Reframer to fine-tune

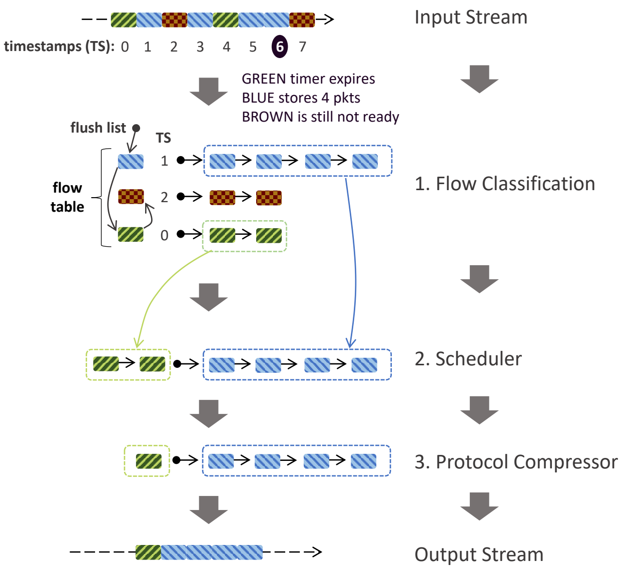


Figure 10: Reframer consists of 3 components: (*i*) a classifier arranges input packets to a flow table, (*ii*) a scheduler flushes flows from the table upon a timeout or burst-size, (*iii*) a compression module coalesces packets to eliminate redundancy.

the buffering time by finding the sweet spot between these two statistics for each application.

Priority. When either the flow classifier informs the scheduler of a full-size batch that is ready to be forwarded or the flush list contains flows reaching the buffering time, the scheduler computes an ordering of the batches based on some configurable *priorities*.

The oldest batches in the flow table are extracted from the head of the flush list. The scheduler resets the per-flow data entries upon emission of the corresponding batches.

In the example shown in Fig. 10, we assume the maximum batch size is 4 packets and the maximum buffer time is equal to 6 time units. At time $t = 7$, Reframer receives the fourth packet of the blue flow and at the same time the buffering time of the green flow expires since the first packet was received at time 0. Both batches are handled by the scheduler for transmission. Reframer's scheduler supports a variety of priority models for ordering batches ready to be sent:

Shortest flow first prioritizes mice over elephant flows.

Oldest flow first prioritizes older over newer flows with respect to the timestamp of the first packet; and

Oldest flow in the queue first prioritizes older over newer flows with respect to their waiting time in the queue.

We envision a tailor-made priority model based upon the network operator's SLOs.

Compressor & Output. Before leaving the Reframer, each batch of packets of the same flow passes through a per-protocol optimizer, e.g., multiple TCP ACKs are coalesced if all packets are in order between ACKs. In the future, we will also look at payload coalescing if the MTU allows it.

Reframer supports an integrated *"bypass"* mechanism. Thus, Reframer allows an operator to define class(es) of traffic that should not be reordered by Reframer based on any given field of the packet (e.g., IP DSCP field). We implemented the obvious case of TCP SYN, as a TCP SYN will never be followed by other packets; therefore, a SYN is never delayed. Additionally, in §5.3 we will show that bypassing mice flows may increase the benefit of Reframer because the possibility of receiving multiple packets of the same mice flow in a period of T_{buff} is low and it is not worth to delay such packets. However, in this work, we have not implemented a heavy hitter detection module, which is left for future optimization and it is not discussed here.

Reframer Implementation. We use FastClick [24] to build a Reframer prototype, which enables many placement scenarios at high speed as will be shown in §5. The classification and flow-state management is handled by its MiddleClick [30] extension, thus the code is only thousand lines long[*].

5 Reframer Evaluation

This section assesses the feasibility and performance of Reframer in increasing the temporal and spatial locality of a stream of traffic by briefly buffering and reordering packets. We evaluate performance at both *per-packet* and *per-flow* granularity in two scenarios: (*i*) to improve the per-packet processing throughput of an NF service chain and (*ii*) to reduce the Flow Completion Time (FCT) of TCP traffic streams served by an HTTP web server. Our results show that the NF chain throughput can be increased by ~84% and the HTTP flow completion times be decreased by 100s of *milliseconds* by simply delaying packets by few 10s or 100s of *microseconds*. Specifically, this section answers the following key questions about the opportunities and challenges in reordering packets: (*i*) *Can Reframer increase the packet processing throughput of an NF chain by increasing the traffic locality of a real-world traffic trace (§5.1)?* (*ii*) *How do the Reframer benefits vary depending on where it is deployed (separate or same server as the application, and then on a CPU core or a SmartNIC (§5.2)?* (*iii*) *How can Reframer handle latency-sensitive traffic (§5.3)?* (*iv*) *Can Reframer reduce the flow completion time of an HTTP web server (§5.4)?*

Testbed. We use the testbed presented in §2.1, running an NF service chain of the NAT and the Firewall presented in §2.2 augmented with a router and a flow statistic NF. First, we place Reframer between the traffic source and the DUT, running on a dedicated Intel Xeon E5-2667 CPU clocked at 2.3 GHz and 128 GB of RAM at 2133 MHz. This machine has two Mellanox ConnectX-6 NICs. While this introduces the cost of a supplementary machine, it gives us an understanding of the *maximum* performance achievable when processing the analysed traffic traces.

[*] The source code is available at: https://github.com/hamidgh09/Reframer

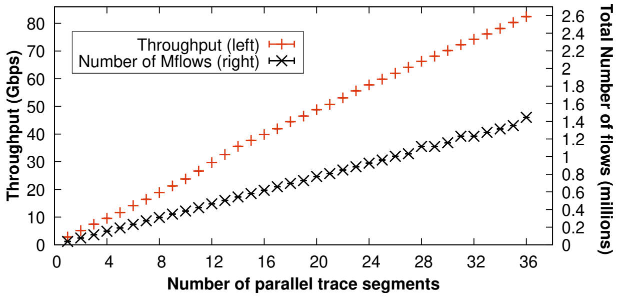


Figure 11: Traces characteristics - the X-axis is the number of multiples of our campus trace played in parallel

Workloads. We use two different types of workloads in our experiments: (i) *per-packet experiments on the NF chain* and (ii) *per-flow experiments with the HTTP web server*. The **per-packet experiments** are based on our campus traffic trace described in §3 with millions of flows in total, a throughput of ~2.2 Gbps, and an average packet size of ~1 KB. To evaluate Reframer with a higher traffic throughput, we split the traffic trace into 32 consecutive windows, each of 20 seconds, and we replay them in parallel from our traffic generator. When splitting the trace, we rewrite the flow identifiers so that any two windows do not have any flow in common (which would otherwise increase the traffic locality of the original trace). Figure 11 shows the number of flows and throughput when running a number of parallel trace segments. For the **per-flow experiment with the HTTP server**, we generate HTTP requests of 1MB files from 4096 clients using WRK [31] towards an NGINX web server.

5.1 Packet-Level Experiments (NF Chain)

In this experiment, we show that (i) Reframer is effective in increasing the spatial traffic locality (i.e., higher *SLF*) of our real-world traffic trace and, consequently, (ii) increasing the throughput of an NF chain. Since the trace is replayed, we focus on *per-packet* metrics (e.g., CPU instructions, latency) and the throughput of the NF chain. The NF chain consists of a Flow Statistic Tracker→Router→Firewall→NAT chain, all implemented in FastClick [24] using state-of-the-art NF elements and DPDK [32] for I/O. We install 10k rules into the firewall and 200 different routes into the router elements. We deploy the chain in a run-to-completion model and we consider it as the *Baseline* in all packet-level experiments. To measure the impact of Reframer, we compare the NFs chain performance *with* and *without* deploying a Reframer instance in front of the chain on an external server. Note that the latency is end-to-end in all the experiments which means it *includes* the time spent in the Reframer buffers. In this experiment, 8 CPU cores are assigned to the NFs chain with 8 RX queues on the NIC (one queue per core). The NIC uses RSS to map traffic among queues. We evaluate alternative deployments with Reframer co-located with the NF chain in §5.2.

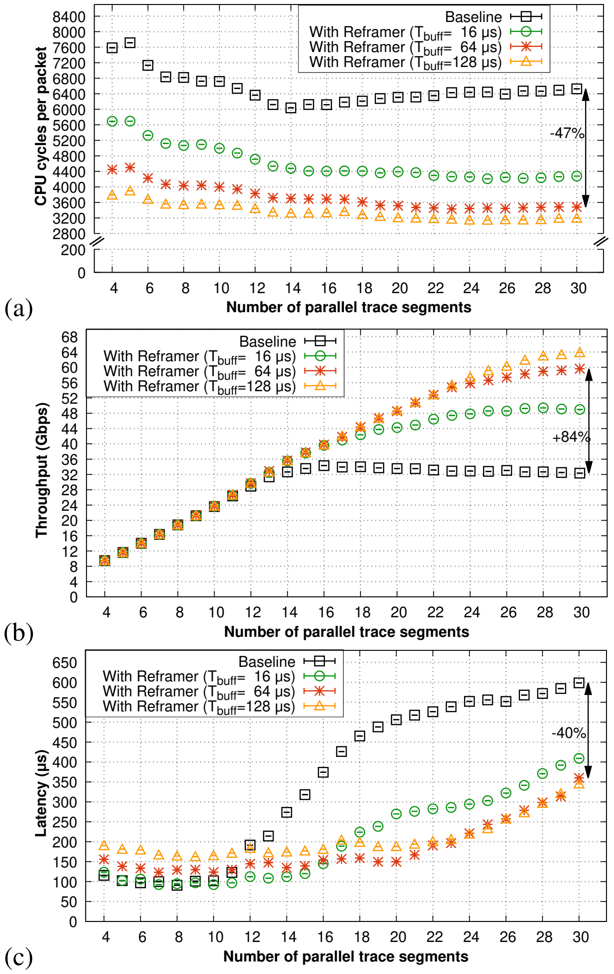


Figure 12: Performance of Reframer versus a baseline NF with increasing load when processing a real trace: (a) CPU cycles per packet, (b) Throughput, and (c) Latency.

Figure 12 shows the effectiveness of Reframer in improving the performance of the NF chain for different workloads (load is expressed as the number of parallel trace segments). At all loads, in Fig. 12(a), we see a substantial decrease in the number of CPU cycles when using Reframer. The reason is the increase in spatial locality from an average of ~1.2, i.e., near the minimum possible spatial locality, to an average of ~1.9, ~2.9, and ~3.3 at the output of the Reframer with 16 μs, 64 μs, and 128 μs of buffering times respectively. Fig. 12(b) shows that at high loads, throughput continues to scale well for T_{buff}= 64 μs and 128 μs, up to ~64 Gbps (a 84-100% improvement) while the throughput peaks at ~48 Gbps for T_{buff}=16 μs. In contrast, the baseline throughput peaks at ~33.6 Gbps and then falls - as the DUT cannot keep up. Fig. 12(c) shows that at low loads, the end-to-end latency is roughly the baseline latency plus T_{buff} when using Reframer. However, we see the Reframer benefit appears as the load increases to maximum capacity of the NFs chain. We discuss and evaluate how to reduce the additional latency introduced by Reframer in §5.3.

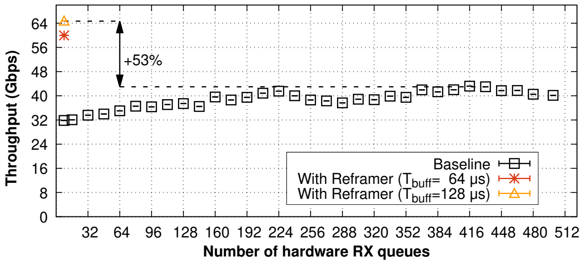
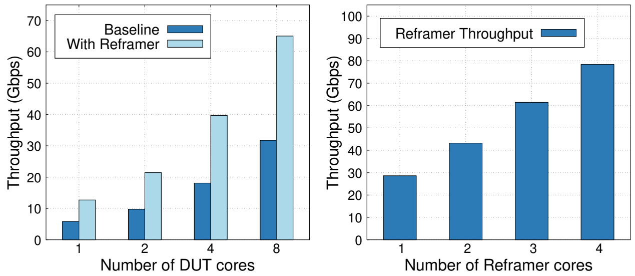


Figure 13: Reframer vs Baseline with various number of hardware RX queues (up to the max. supported by the NIC).



(a) DUT throughput.

(b) Reframer capacity.

Figure 14: Maximum throughput of Reframer and DUT with different number of cores.

High number of RX queues has small impact on the DUT throughput. We repeated the above experiment with 30 parallel trace segments and various numbers of RX queues on the DUT in a range of 8 to 500 (which is the maximum possible number of RX queues on the DUT's NIC). We preserve the total number of descriptors around 8192 by setting per queue descriptors to $max(32, 2^{14-\lceil \log N \rceil})$ where N is the total number of RX queues. In this experiment we show that by increasing the number of RX queues the average spatial locality increases from ~1.2 to ~2.5 without Reframer. However, despite the improvement in the traffic locality, Figure 13 shows only a slight increase in the maximum throughput of baseline. The main reason is, having hundreds of RX queues leads to more empty polling in the DUT which is costly and negatively affects the performance. It is worth noting that, fetching incoming packets from RX queues is hardware-specific and depends on the data structures that NICs are using to process incoming packets; hence, optimizing algorithms and data structures in future NICs may lead to better results. However, discussing the future road map of NICs is out of scope of this paper. On the other hand, when Reframer is located between the traffic source and DUT, increasing the number of DUT RX queues has a negative impact on the throughput because incoming packets are already sorted and classifying flows in different hardware queues does not increase packets' locality. So we set 8 RX queues (one per core) for DUT when Reframer exists in the network. Finally, we see 53% more throughput with Reframer vs. using hundreds of RX queues for the baseline case.

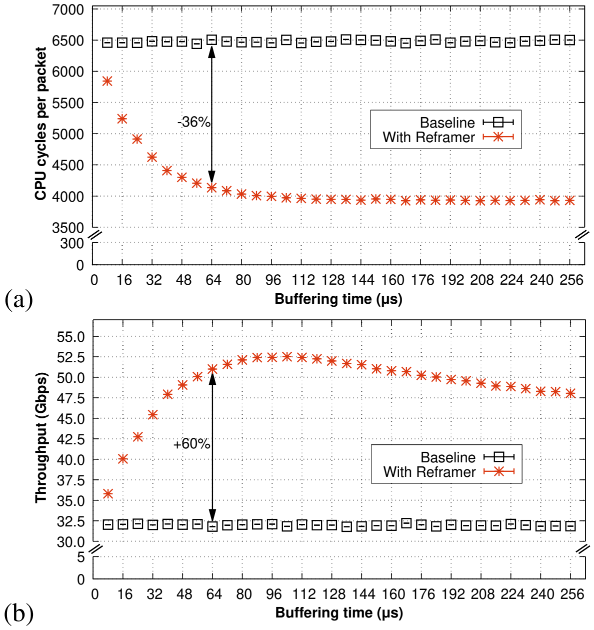


Figure 15: Impacts of Reframer when collocated with the NF chain: (a) Cycles per packet and (b) Throughput.

Packets' locality benefit persists with various number of DUT cores. We also show that Reframer benefits do not depend on the number of DUT cores. To do so, we measured the maximum throughput of DUT by running the experiment with various number of cores assigned to DUT. Reframer's buffering time is set to 128 µs in all cases. Figure 14a demonstrates that the throughput increase rate is almost the same for different number of cores.

Reframer scales almost linearly with the number of cores. As we discussed in §4, Reframer benefits from an optimized data structure to classify, order, and flush packets in a constant time. Our stress test reveals that Reframer is able to handle up to 28 Gbps with only *one core*. Here, we increase the offered load gradually until we see ~1% packet drops in Reframer. Figure 14b shows that Reframer's capacity increases *almost linearly* when increasing the the number of cores.

5.2 Same-Server Deployment

In the previous section, we showed that deploying Reframer on a dedicated server increases spatial and temporal locality, ultimately resulting in significant performance gains. In the following experiments, we evaluate deploying Reframer on the same server where an application is running. Using the same NF chain (Baseline) as previously, we consider two deployments: (i) chaining Reframer with the NF chain, i.e., the entire chain running to completion on the same CPU cores (referred to as *in-chain* deployment), and (ii) deploying Reframer on a SmartNIC.

In-chain deployment. We evaluate the performance of Re-

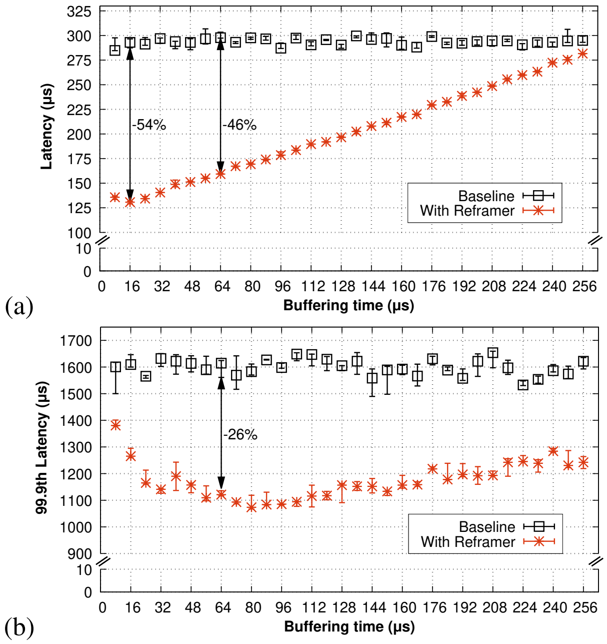


Figure 16: Impacts of Reframer when collocated with the NF chain: (a) Average latency and (b) 99.9th percentile latency.

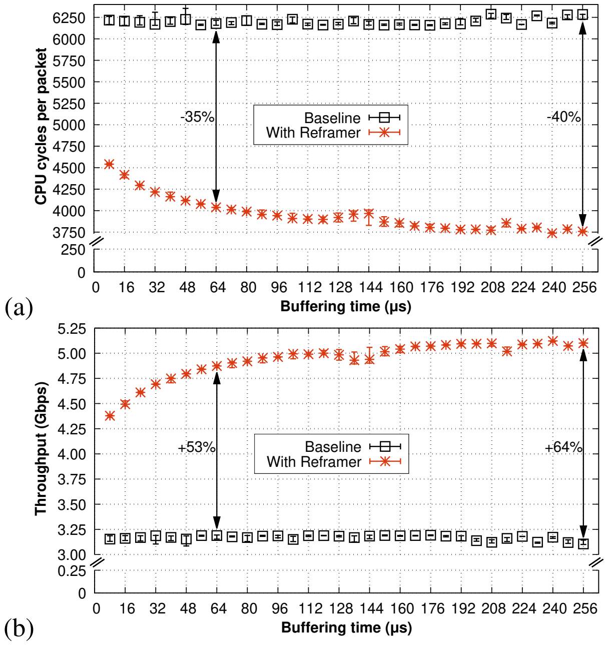


Figure 17: Impact of Reframer when offloaded into a Smart NIC which precedes the NF chain: (a) Cycles per packet, (b) Throughput. Latency is given in Appendix A.2

framer for the *in-chain* deployment versus the *baseline* for different buffering times. Generally, increasing buffering time in Reframer will lead to more packet locality, since it increases the possibility of receiving more packets of the same flow; Hence, we see a considerable increase in the DUT throughput and reduction in the end-to-end latency. Fig. 15 shows that by placing Reframer right before the service chain, the number of cycles per packet decreases with increasing buffering time while throughput increases by 60% when Reframer buffers packets for 64 µs. To evaluate the impact of Reframer on the packets end-to-end latency, we restrict the incoming packet rate to ~30 Gbps which is less than the maximum capacity of DUT in the baseline mode. In Fig. 16 we can see the average latency is reduced by 46% with T_{buff}=64 µs. Additionally, Reframer improves the tail latency by ~26% even when it is collocated with service chain on the same server. In this experiment, latency benefits start to fade gradually from a specific buffering time because the cost of delaying packets surpasses the processing speed-up. The baseline numbers are mostly the same for all x axis values because we have no buffering in baseline mode. The fluctuation in baseline values is inevitable because DUT cores are at a maximum load.

SmartNIC deployment. As a proof-of-concept deployment for offloading Reframer into a NIC to save CPU core resources on the server, we deployed Reframer on two ARM cores of a Mellanox Bluefield SmartNIC – equipped with 16×64-bit Armv8 A72 cores and two 100 Gbps ports while the NFs chain works on a single CPU core. Fig. 17 shows improvements in throughput similar to the in-chain deployment. We discovered that the performance using a single ARM core was limited by the current Mellanox drivers for the cards, a constraint/limitation confirmed with Mellanox.

Flow-oblivious batching is highly suboptimal. We also compare Reframer with a Batchy-like [14] implementation written in FastClick. Batchy is a state-of-the-art packet processing system that buffers packets in a *flow-oblivious* manner at multiple locations in an NF chain, i.e., Batchy does not create bursts of packets from the same flow but mix all flows that must be processed by the same NF element. We observe that Batchy improves the throughput of the chained NFs by 4%, whereas Reframer improves throughput by 48%. These results corroborate our analysis in section (§2), where we showed how detrimental it is to process streams of packets that are highly interleaved between different flows as opposed to per-flow batches.

5.3 Latency-Sensitive Flows

In our previous experiments, Reframer delayed all types of packets for a T_{buff} interval, possibly increasing the FCT or packet processing time of short flows. We argue that an operator could explicitly tag which traffic classes should be delayed to improve application throughput. To evaluate the impact of delaying only large flows, we ran an experiment similar to the one described in §5.1, but we explicitly flag only large flows so that Reframer can batch them while bypassing the unflagged packets and show the results for increasing number of parallel trace segments in Fig. 18. Compared to the case where all flows are delayed, the throughput of

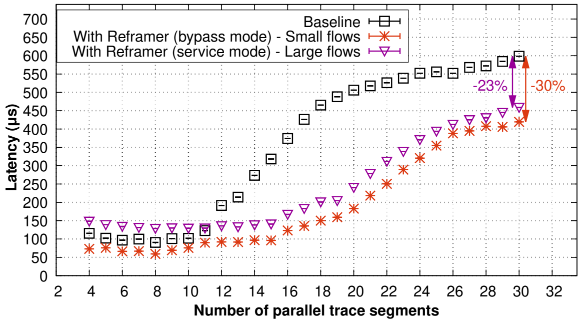


Figure 18: Reframer provides differentiated services by prioritizing small flows over large flows which are bypassed.

the NF chain slightly decreases (between 0% and 4% according to the number of parallel trace segments), while the latency of the packets belonging to the small flows align with the best of the baseline (at low loads) or the latency of Reframer minus the buffering delay (at higher loads). Somewhat surprisingly, Reframer achieves *lower latencies* than the baseline for small flows across *all* traffic loads by simply delaying and reordering only large flows. We leave the detection of heavy-hitters, for instance detecting which flows could have generated multiple larger bursts, as future work.

5.4 Flow-Level Experiments (HTTP Server)

In this experiment, we evaluate Reframer to assess its impact on a web server application using TCP connections to dispatch files of 1MB to a set of 2048 clients continuously fetching files. By controlling the rate and number of clients' requests, we are also able to substantially increase the throughput of the test and exploit the 100G NIC interfaces. To simulate 4096 independent clients with more realistic latencies, we place a machine in-the-wire that delays packets in per-flow queues by ~10 ms±~2 ms. Hence, each connection exhibits slightly different delays, for an average ~20 ms delay. The focus of this experiment is on flow-level metrics, with the goal to check whether (*i*) Reframer improves the FCT of the dispatched files and (*ii*) the buffering delays cause any troubles to the underlying congestion control mechanism (i.e., TCP Cubic). We compare the *baseline* against Reframer. We selected NGNIX 1.14 as the web server running on 16 cores of the DUT, while Reframer runs on a dedicated NF machine using 6 cores. Reframer reorders packets in both directions, aggregates TCP ACKs from the client to the server, and eventually reorders out-of-order TCP packets. Fig. 19(a) shows that Reframer increases the application throughput by 20%. The observed improvements are due to the increase in spatial locality from 1.25 to 14. Fig. 19(b) shows that despite introducing delays in the order of microseconds, Reframer *reduces* FCT of TCP connections by fractions of a second (from 3.4 s to 3.1 s). ACK coalescing accounts for ¼ of the throughput improvements but does not affect the FCT.

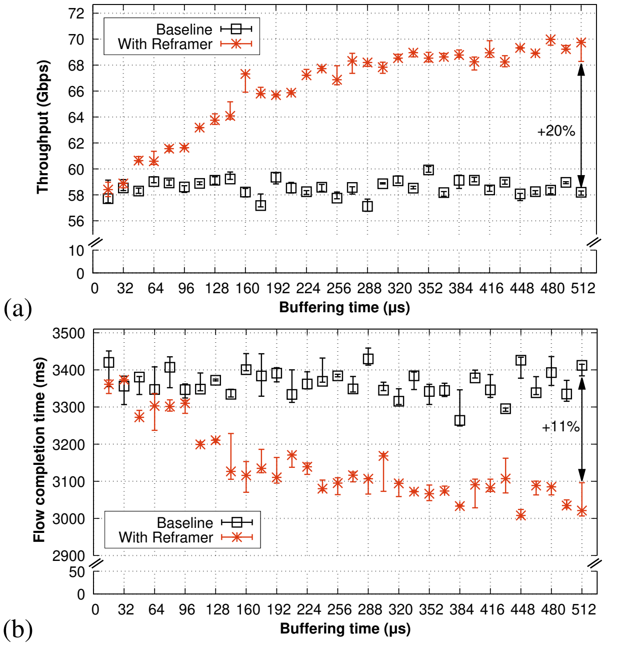


Figure 19: Impact of Reframer when reordering packets of HTTP flows: (a) Throughput and (b) FCT.

6 Related Work

Batching. Previous efforts [14, 24, 33–35] have shown the importance of processing entire batches of packets rather than individual packets (not necessarily belonging to the same flow) in order to amortize the costs of the interrupts in the NF processing system (e.g., Batchy [14], SCC [35]). Our work is orthogonal to these approaches because Reframer improves the performance of a server application in a "transparent" way, e.g., by reordering packets on the NIC or before being sent to the application. Moreover, existing packet processors do not increase the traffic locality at the per-flow level, which we show to be critical to achieve high performance in §5.2.

Traffic coalescing. Receive Side Coalescing (RSC) [36] aka LRO accelerates TCP processing by merging consecutive packets of a TCP flow into a single frame. Unfortunately, as shown in §2.2, hardware-based LRO breaks as soon as packets are interleaved. Similarly, the software implementation of LRO in the Linux kernel, called Generic Receive Offload (GRO) [37], suffers from the same problem.

Packet schedulers. We distinguish between *hardware* and *software* packet schedulers. Hardware packet schedulers typically try to realize different approximations of universal schedulers mapping packet ranks to the available queues on the hardware (e.g., [38–43]). Another set of hardware packet schedulers (e.g., [44–52]) focus on network-level optimization in datacenters (e.g., minimize traffic congestion). None of *all* these works have explicitly looked at the possibility of batching and scheduling packets to increase traffic locality at the per-flow level. For instance, pFabric [44] would *nullify* any high traffic locality by purposely interleaving flows.

Stardust [53] is a hardware-based fabric architecture for datacenter scale networks. Stardust classifies packets per destination and chops them into bounded-size cells. This technique enables Stardust to send chops of packets in a burst, which potentially minimizes the cost of processing them at the destination. However, based on our understanding, Stardust's cells are flow-agnostic, whereas Reframer improves the performance of NFs and applications by increasing the traffic locality at the per-flow level. Moreover, Reframer *purposely* delays packets, which is the *pivotal* aspect of our proposed solution.

Software-based packet schedulers (e.g., [30, 54–61]) operate at the CPU level with the goal of dispatching the incoming flows (or coflows) of traffic to the different cores on the machine running packet processing operations. Also in this case, *none* of these approaches have explicitly looked at the impact of traffic locality on the performance of the applications receiving the packets.

To summarize, existing scheduling schemes do *not* take into consideration the impact of per-flow traffic locality. In contrast, Reframer schedules and prioritizes bursts of flows using a variety of policies while exploiting opportunities for packet coalescing and increased traffic locality.

TCP accelerations. AccelTCP [62] and Tonic [63] are dual-stack solutions that offload or generalize stateful TCP operations to NICs in order to simplify the end host stack. Such operations include connection setup and teardown as well as connection splicing that relays packets of two connections entirely within a NIC. To the best of our knowledge, *none* of these works explicitly aim to increase per-flow traffic locality.

7 Conclusions

This work unveiled the importance of packet ordering on many applications, specifically NFs and TCP applications. We showed that receiving traffic by bursts of packets of the same flow could improve a server's performance by a factor of 3× as opposed to receiving packets of interleaved flows. Analyzing realistic traffic, we found that by slightly delaying traffic, even by only 64 µs, one can potentially re-build bursts of packets. We then described Reframer, a software NF, capable of re-building bursts at 28 Gbps with a single core, and scalable to 100 Gbps with a few cores. We showed Reframer is still highly beneficial when deployed as part of an NF chain, while bringing performance improvements to the server and its services. We believe this paper will spur new research around the delicate interaction between congestion control mechanisms and cache-based optimizations. It also calls for further potential improvements, e.g., decreasing the number of frames by coalescing payload or realizing Reframer in hardware.

Acknowledgements

We would like to thank our shepherd Mahesh Balakrishnan and the anonymous reviewers for their insightful comments and suggestions on this paper. This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 770889). It was also funded by the Swedish Foundation for Strategic Research (SSF). The work was partially supported by KTH Digital Futures and the Wallenberg AI, Autonomous Systems, and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[1] Intel Barefoot Networks. Tofino-2 Second-generation of World's fastest P4-programmable Ethernet switch ASICs, 2020. https://www.barefootnetworks.com/products/brief-tofino-2/.

[2] NVIDIA Mellanox. ConnectX®-6 EN IC 200GbE Ethernet Adapter IC, 2019. https://www.mellanox.com/related-docs/prod_silicon/PB_ConnectX-6_EN_IC.pdf.

[3] Zhiping Yao, Jasmeet Bagga, Hany Morsy. Introducing Backpack: Our second-generation modular open switch, November 2016. https://engineering.fb.com/data-center-engineering/introducing-backpack-our-second-generation-modular-open-switch/.

[4] Shelby Thomas, Rob McGuinness, Geoffrey M. Voelker, and George Porter. Dark packets and the end of network scaling. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, ANCS '18, page 1–14, New York, NY, USA, 2018. Association for Computing Machinery.

[5] Shelby Thomas, Geoffrey M. Voelker, and George Porter. Cachecloud: Towards speed-of-light datacenter communication. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, Boston, MA, July 2018. USENIX Association.

[6] NVIDIA Mellanox Technologies. RDMA and RoCE for Ethernet Network Efficiency Performance, 2020. https://www.mellanox.com/products/adapter-ethernet-SW/RDMA-RoCE-Ethernet-Network-Efficiency.

[7] Intel. Data Direct I/O Technology, 2017. http://www.intel.co.jp/content/dam/www/public/us/en/documents/white-papers/data-direct-i-o-technology-overview-paper.pdf.

[8] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 673–689. USENIX Association, July 2020.

[9] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Make the Most out of Last Level Cache in Intel Processors. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 8:1–8:17, New York, NY, USA, 2019. ACM. http://doi.acm.org/10.1145/3302424.3303977.

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

[11] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. PacketMill: Toward per-core 100-Gbps Networking. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, New York, NY, USA, 2021. Association for Computing Machinery.

[12] Yifan Yuan, Yipeng Wang, Ren Wang, and Jian Huang. HALO: Accelerating Flow Classification for Scalable Packet Processing in NFV. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 601–614, New York, NY, USA, 2019. ACM. http://doi.acm.org/10.1145/3307650.3322272.

[13] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *ACM Queue*, 14, September-October:20 – 53, 2016.

[14] Tamás Lévai, Felicián Németh, Barath Raghavan, and Gabor Retvari. Batchy: Batch-scheduling Data Flow Graphs with Service-level Objectives. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 633–649, Santa Clara, CA, February 2020. USENIX Association. https://www.usenix.org/conference/nsdi20/presentation/levai.

[15] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/.

[16] Open vSwitch. An Open Virtual Switch. http://openvswitch.org.

[17] VMware. NSX-T Data Center Documentation, 2020. https://docs.vmware.com/en/VMware-NSX-T-Data-Center/index.html.

[18] OpenStack. Open Source Cloud Computing Software, 2020. https://www.openstack.org/.

[19] Red Hat. OpenShift - The Kubernetes platform for big ideas, 2020. https://www.openshift.com/.

[20] The Linux Foundation. Kubernetes, 2020. https://kubernetes.io/.

[21] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. The Design and Implementation of Open vSwitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 117–130, Berkeley, CA, USA, 2015. USENIX Association. https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-pfaff.pdf.

[22] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification Using Tuple Space Search. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, page 135–146, New York, NY, USA, 1999. Association for Computing Machinery. https://doi.org/10.1145/316188.316216.

[23] Andrew Theurer, Red Hat. Testing the Performance Impact of the Exact Match Cache. In *Open vSwitch 2018 Fall Conference*, pages 1–17, San Jose, CA, USA, December 2018. https://www.openvswitch.org/support/ovscon2018/5/1330-theurer.pdf.

[24] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 5–16, Washington, DC, USA, May 2015. IEEE Computer Society. http://dl.acm.org/citation.cfm?id=2772722.2772727.

[25] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: congestion-based congestion control. *ACM Queue*, 14(5):20–53, 2016. http://doi.acm.org/10.1145/3012426.3022184.

[26] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *SIGCOMM Comput. Commun. Rev.*,

45(4):537–550, August 2015. https://doi.org/10.1145/2829988.2787510.

[27] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 19–19, Berkeley, CA, USA, 2012. USENIX Association. http://dl.acm.org/citation.cfm?id=2228298.2228324.

[28] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 404–417, New York, NY, USA, 2017. ACM. http://doi.acm.org/10.1145/3098822.3098852.

[29] Allen B. Downey. TCP Self-Clocking and Bandwidth Sharing. *Comput. Netw.*, 51(13):3844–3863, September 2007. https://doi.org/10.1016/j.comnet.2007.04.005.

[30] Tom Barbette, Cyril Soldani, Romain Gaillard, and Laurent Mathy. Building a chain of high-speed VNFs in no time. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, Bucharest, Romania, June 2018. IEEE. https://doi.org/10.1109/HPSR.2018.8850742.

[31] Will Glozer. wrk - a HTTP benchmarking tool. https://github.com/wg/wrk.

[32] Linux Foundation. Data Plane Development Kit (DPDK), 2020. http://www.dpdk.org.

[33] Luigi Rizzo. netmap: A novel framework for fast packet i/o. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2012. http://info.iet.unipi.it/~luigi/netmap/.

[34] Joongi Kim, Seonggu Huh, Keon Jang, KyoungSoo Park, and Sue Moon. The power of batching in the Click modular router. In *Proceedings of the ACM Asia-Pacific Workshop on Systems (APSYS)*, 2012. http://doi.acm.org/10.1145/2349896.2349910.

[35] Georgios P. Katsikas, Gerald Q. Maguire Jr., and Dejan Kostić. Profiling and accelerating commodity NFV service chains with SCC. *Journal of Systems and Software*, 127C:12–27, February 2017. https://doi.org/10.1016/j.jss.2017.01.005.

[36] Srihari Makineni, Ravi Iyer, Partha Sarangam, Donald Newell, Li Zhao, Ramesh Illikkal, and Jaideep Moses. Receive Side Coalescing for Accelerating TCP/IP Processing. In *Proceedings of the 13th International Conference on High Performance Computing*, HiPC'06, pages 289–300, Berlin, Heidelberg, 2006. Springer-Verlag. http://dx.doi.org/10.1007/11945918_31.

[37] Jonathan Corbet. Generic receive offload, 2009. https://lwn.net/Articles/358910/.

[38] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable Packet Scheduling at Line Rate. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 44–57, New York, NY, USA, 2016. ACM. http://doi.acm.org/10.1145/2934872.2934899.

[39] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Universal Packet Scheduling. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation*, NSDI'16, pages 501–521, Berkeley, CA, USA, 2016. USENIX Association. http://dl.acm.org/citation.cfm?id=2930611.2930644.

[40] Vishal Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 367–379, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3341302.3342090.

[41] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 59–76, Santa Clara, CA, February 2020. USENIX Association. https://www.usenix.org/conference/nsdi20/presentation/alcoz.

[42] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable Calendar Queues for High-speed Packet Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 685–699, Santa Clara, CA, February 2020. USENIX Association. https://www.usenix.org/conference/nsdi20/presentation/sharma.

[43] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and Efficient NIC Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages

33–46, Boston, MA, February 2019. USENIX Association. https://www.usenix.org/conference/nsdi19/presentation/stephens.

[44] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. PFabric: Minimal near-Optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 435–446, New York, NY, USA, 2013. Association for Computing Machinery. https://doi.org/10.1145/2486001.2486031.

[45] Y. Lu, G. Chen, L. Luo, K. Tan, Y. Xiong, X. Wang, and E. Chen. One more queue is enough: Minimizing flow completion time with explicit priority notification. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017. https://doi.org/10.1109/INFOCOM.2017.8056946.

[46] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 50–61, New York, NY, USA, 2011. Association for Computing Machinery. https://doi.org/10.1145/2018436.2018443.

[47] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 307–318, New York, NY, USA, 2014. Association for Computing Machinery. https://doi.org/10.1145/2619239.2626309.

[48] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. PHost: Distributed near-Optimal Datacenter Transport over Commodity Network Fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2716281.2836086.

[49] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3230543.3230564.

[50] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient Coflow Scheduling with Varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 443–454, New York, NY, USA, 2014. Association for Computing Machinery. https://doi.org/10.1145/2619239.2626315.

[51] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. Sincronia: Near-Optimal Network Design for Coflows. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 16–29, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3230543.3230569.

[52] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 455–468, USA, 2015. USENIX Association.

[53] Noa Zilberman, Gabi Bracha, and Golan Schzukin. Stardust: Divide and conquer in the data center network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 141–160, 2019.

[54] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. Eiffel: Efficient and Flexible Software Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 17–32, Boston, MA, February 2019. USENIX Association. https://www.usenix.org/conference/nsdi19/presentation/saeed.

[55] Vojislav Đukić, Sangeetha Abdu Jyothi, Bojan Karlas, Muhsen Owaida, Ce Zhang, and Ankit Singla. Is advance knowledge of flow sizes a plausible assumption? In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 565–580, Boston, MA, February 2019. USENIX Association. https://www.usenix.org/conference/nsdi19/presentation/dukic.

[56] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 71–84, New York, NY, USA, 2017. ACM. http://doi.acm.org/10.1145/3098822.3098828.

[57] Guikai Liu and Qing Li. Fair and Efficient Packet Scheduling Using Resilient Quantum Round-Robin. *Journal of Networks*, 9(2):269–276, 2014. https://doi.org/10.4304/jnw.9.2.269-276.

[58] Anat Bremler-Barr, Yotam Harchol, and David Hay. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 511–524, New York, NY, USA, 2016. ACM. http://doi.acm.org/10.1145/2934872.2934875.

[59] Georgios P. Katsikas, Marcel Enguehard, Maciej Kuźniar, Gerald Q. Maguire Jr., and Dejan Kostić. SNF: Synthesizing high performance NFV service chains. *PeerJ Computer Science*, 2:e98, November 2016. http://dx.doi.org/10.7717/peerj-cs.98.

[60] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, Jr., and Dejan Kostić. RSS++: load and state-aware receive side scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, pages 318–333, New York, NY, USA, 2019. ACM. http://doi.acm.org/10.1145/3359989.3365412.

[61] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Gerald Q. Maguire Jr., and Rebecca Steinert. Metron: High-Performance NFV Service Chaining Even in the Presence of Blackboxes. *ACM Trans. Comput. Syst.*, 38(1–2), July 2021. https://doi.org/10.1145/3465628.

[62] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 77–92, Santa Clara, CA, February 2020. USENIX Association. https://www.usenix.org/conference/nsdi20/presentation/moon.

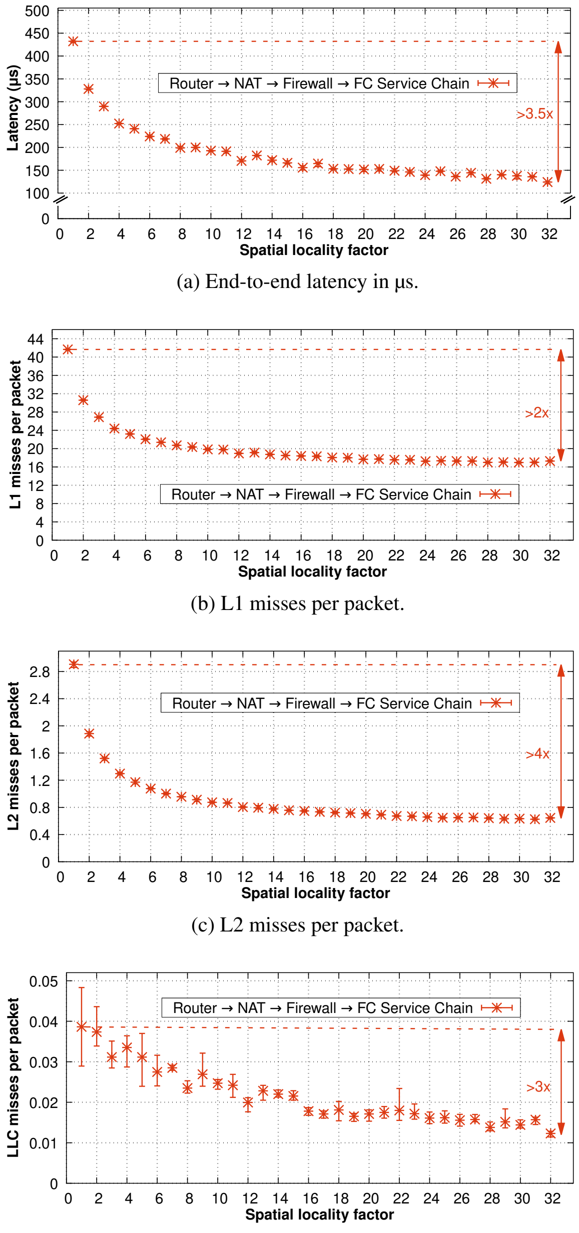
[63] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed nics. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 93–109. USENIX Association, 2020.

A Supplementary Material

This section provides some additional material for this paper.

A.1 Deploying a chain of NFs

In addition to experiments discussed in §2.4, we deployed a chain of network functions on the DUT as a complementary experiment. In this test, we connected a Router, a NAT, a firewall, and a Flow statistics counter (FC) in a row, as a chain of NFs. The DUT uses 4 CPU cores to serve the packets and it is implemented in a run-to-completion model to exploit the parallelism on the processors. All the other configurations are similar to §2.4.



(a) End-to-end latency in µs.

(b) L1 misses per packet.

(c) L2 misses per packet.

(d) LLC misses per packet.

Figure 20: Impact of packet order on the performance of a Router→NAT→Firewall→FC chain of NFs.

Since the deployed chain is both CPU and memory intensive, the scale of CPU cycles per packet and the end-to-end latency are higher in compare to individual NAT and firewall experiments in §2.4. However, the results in Figure 20 confirm that, regardless of the complexity of the implemented

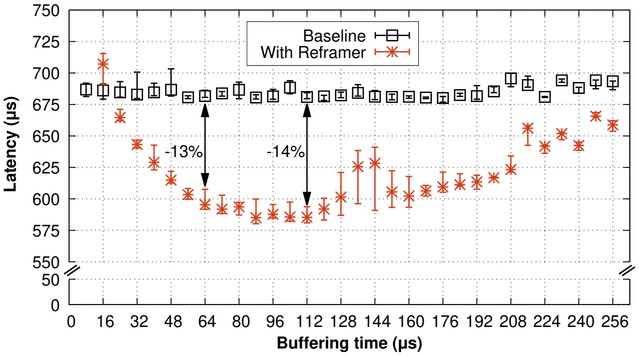


Figure 21: Latency of Reframer when offloaded into a Smart NIC which precedes the NF chain.

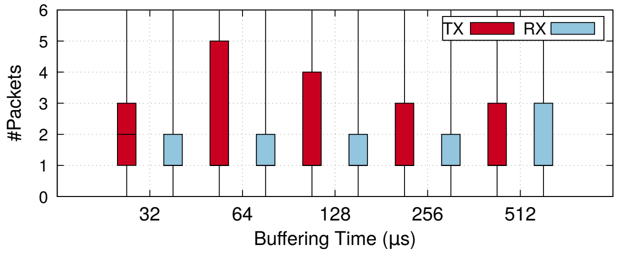


Figure 22: Impact of increasing the waiting time on the probability of coalescing Transmission Control Protocol (TCP) ACKs. (The figure is intentionally scaled to enhance the visibility of the first to third quartiles.)

NFs, ordering the packets has a significant impact on the DUT's performance.

Similar to §2.4, the fundamental reason of such enhancement is efficient utilization of system caches. In this experiment, since the DUT needs more data to process a packet, the improvement in cache misses has been extended to L2 and LLC. Figure 20d shows a substantial improvement in terms of number of LLC misses. Note that this improvement is not happening only in LLC. We also can see the same trend (with smaller improvement factors) in L1 (Figure 20b) and L2 (Figure 20c) caches.

A.2 Running Reframer in a SmartNIC

Figure 21 shows the latency induced by the Reframer versus a baseline NF, when deployed on two Arm cores of a Mellanox Bluefield SmartNIC.

A.3 Analyzing the Trace

To perform the analysis, we have used PcapPlusPlus to create a CSV file composed of useful fields. Then, we split the 62-GB file to per-flow CSV files via Spark. Finally, we use Python data science libraries (e.g., Pandas and NumPy) to calculate the probability of receiving different batch sizes within different time windows. Listing 1 shows the python code used to process each flow.

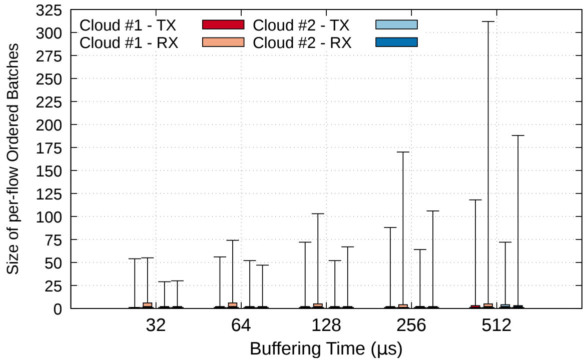


Figure 23: Impact of increasing the waiting time on the probability of receiving packets with the same TCP flow.

ACK coalescing. Fig. 22 shows the potential improvement from TCP ACK coalescing in the campus trace. We calculated the distribution of the number of per-flow packets with enabled TCP acknowledgment flag (ACK) within a time frame.

```python
import pandas as pd
import numpy as np

# Calculate the size of ordered batches for each flow
# based on the input window size (ws)
def process(flow,ws):
    # Getting the timestamp of packets in a flow
    ts = flow['ts'].to_numpy()

    # Initialize variables
    batch_size=1
    i=1
    ordered_size=np.empty((0))
    threshold=32

    # Check the size of flow
    if ts.size == 1:
        # Add the batch_size to the array of ordered_size
        ordered_size = np.append(ordered_size,batch_size)
    else:
        # Sort the timestamps
        sorted_ts = np.sort(ts)
        # Start from the first packet of a flow
        base_ts = sorted_ts[0]

        # Continue while there is still more packets
        while i < sorted_ts.size:

            # Increase the size as long as the next packet
            # arrives before the end of the window size
            if sorted_ts[i] - base_ts < ws :
                batch_size = batch_size + 1

            # If the size of the batch is larger than
            # the threshold or the next packet of the flow
            # comes after the end of the window size.
            # The batch size and the time counter,
            # we should stop the time counter. Stopping
            # the counter means that we start the counter
            # again with the next packet. Also, we update the
            # beginning of the window size with the timestamp
            # of the newly arrived packet.
            if (batch_size >=threshold) \
                or (sorted_ts[i] - base_ts >= ws):
                # Update the beginning of the window size
                base_ts = sorted_ts[i]
                # Add the batch size to the array
                ordered_size = np.append(ordered_size,batch_size)

                # Reset the batch size
                if batch_size != 1:
                    batch_size = 1
                    i = i + 1
                    ordered_size = np.append(ordered_size,count)
    return ordered_size
```

Listing 1: Python function used to calculate the size of the per-flow batches.

Table 1: Flow statistics per Internet Protocol (IP) address of two popular cloud providers.

IP		#Flows		Flow Size (#Packets)							
				Min		Mean		Median		Max	
		TX	RX	TX	RX	TX	RX	TX	RX	TX	RX
Cloud-1	IP-1	19985	20039	1	1	47.45	54.08	16	16	87131	221594
	IP-2	5384	5433	1	1	14.92	19.11	13	15	92	157
	IP-3	4741	4748	1	1	42.55	51.58	15	15	43500	32540
	IP-4	4567	4564	1	1	52.62	37.18	16	16	38515	19168
	IP-5	4245	3805	1	1	187.12	1397.09	8	29	57392	217309
	IP-6	4155	4000	1	1	12.83	13.63	12	10	831	2775
	IP-7	3980	3958	1	1	13.48	9.63	11	8	663	394
	IP-8	3759	3759	2	2	258.30	318.38	11	10	33403	33356
	IP-9	3154	3159	1	1	28.86	21.89	14	10	310	279
	IP-10	2996	2984	1	1	39.76	22.48	56	28	238	228
Cloud-2	IP-1	19776	19762	1	1	165.77	137.07	10	10	1615124	764636
	IP-2	18120	18103	1	1	19.26	44.42	10	10	17929	42637
	IP-3	15967	15945	1	1	39.33	68.71	11	11	68306	92210
	IP-4	11168	11150	1	1	105.09	62.86	10	10	255327	121520
	IP-5	9207	9235	1	1	110.75	119.08	21	20	9129	7665
	IP-6	8828	8803	1	1	521.57	265.83	10	10	1353933	587496
	IP-7	5897	5879	1	1	51.44	67.54	15	14	12448	13422
	IP-8	5330	4993	1	1	42.93	77.05	12	11	7248	18137
	IP-9	4499	4479	1	1	116.08	198.77	16	15	16625	37459
	IP-10	3785	3775	1	1	57.22	75.43	17	16	4371	8287
	IP-11	3369	3362	1	1	235.40	306.51	17	15	19614	22311
	IP-12	3355	3279	1	1	1501.01	493.11	18	19	4905771	1409104
	IP-13	3152	3144	1	1	23.20	33.87	11	10	867	2617
	IP-14	3113	3078	1	1	28.69	47.82	15	14	2952	7922
	IP-15	2864	2868	1	1	59.26	83.18	12	12	9143	22045