GranularNF: Granular Decomposition of Stateful NFV at 100 Gbps Line Speed and Beyond

Ziyan Wu, Tianming Cui, Arvind Narayanan, Yang Zhang,
Kangjie Lu, Antonia Zhai, Zhi-Li Zhang
University of Minnesota – Twin Cities

ABSTRACT

In this paper, we consider the challenges that arise from the need to scale virtualized network functions (VNFs) at 100 Gbps line speed and beyond. Traditional VNF designs are monolithic in state management and scheduling: internally maintaining all states and operations associated with them. Without proper design considerations, it suffers from limitations when scaling at 100 Gbps link speed and beyond: the inability of efficient utilization of the cache because of the contention due to the frequent control plane activities, computational/memory-intensive tasks taking up CPU times, shares states causing the synchronization among the cores.

We address these limitations by arguing for the need to **granularly** decompose a VNF into data/control components that are co-located within a server but can be independently scaled among the cores. To realize the approach, we design a "serverless" programming framework with novel abstraction to optimize the data components that must process packets at the line speed, reduce the contention of the data states and enable run-time scheduling of different components for improved resource utilization. The abstractions, combined with the runtime system that we design, help NFV developers focus on the logic and correctness of VNF programming without worrying about how VNFs may be scaled in or out. We evaluate our platform by comparing it with monolithic approaches using different workloads and by analyzing its advantages of separation on scalability, performance determinism, and feature velocity.

1 INTRODUCTION

Unlike traditional data centers, edge clouds have limited real estate (e.g., rack space): packing more networking, e.g., with 100 Gbps network interface cards (NICs) and more cores per server is crucial. Network Function Virtualization (NFV) systems, e.g., implementing 5G core network functions, running in edge clouds therefore must be capable of processing packets in software at 100 Gbps line speed and beyond.

VNF designs (see, e.g., [9, 17, 18]) often apply the SBA (server-based architecture), where it decouples the high-level policymaking into a separate network of network functions using the idea of CUPS (Control and User Plane Separation) and lets the data network functions communicate with the

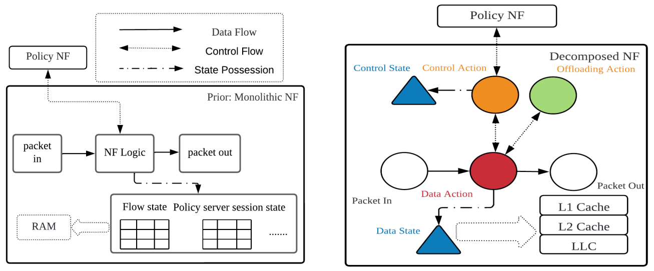


Figure 1: (left) Monolithic NF; (right) Granularly Decomposed NF

control network function through a control message channel. The architecture improves the flexibility of NF software but the principle is not sufficient addressing the issue of scaling stateful network function within a multi-core server. Existing programming abstractions [6, 8, 15] focus on the reusability of packet processing modules. They compose different building blocks or elements into more complex network functions and states, exploiting the reusability of different modules to increase performance (Figure 1 left).

While previous works show considerable improvement combined with a fast packet processing framework [4], there are limitations when achieving line speed. Without additional design considerations, it suffers from limitations when scaling at 100 Gbps link speed and beyond. First, the utilization of the cache often suffers from interference because they do not differentiate states/actions with different properties. As a result, different states often contend with each other whereas the states that highly impact the performance do not reside in the cache. Second, there is a lack of proper scheduling of routines that perform computational/memory-intensive tasks. Scaling out often treats NF as a whole with coarse resource allocation, regardless of the real bottlenecks.

For the efficient execution of a single stateful network function on an NFV system, we argue that in order to scale them effectively on a multi-core system, we need to decompose it into components with different properties. The **granularly decomposed NF** (Figure 1 right) is an organization of independently schedulable data components (data actions and data states), control components (control actions and control states), and offloading components, which communicate with

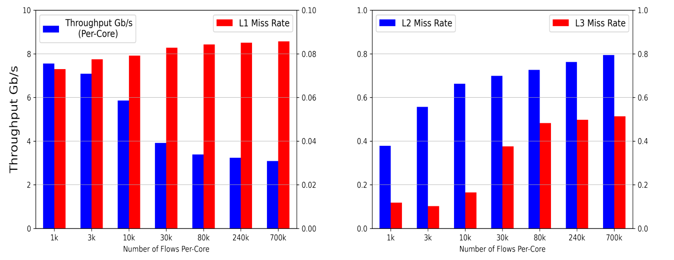


Figure 2: (left) The Size of the Working Set of Instances; (right) Impacts Performance for Stateful Load Balancers

each other through message passing. The data components are responsible for per-packet processing, the control components for policy-related computation and communication with the remote policy server, and the offloading components for CPU-intensive computation or memory-intensive tasks.

Compared to the monolithic NFs, the benefits of granularly decomposed ones not only inherit the benefits from a serverless framework such as independent scaling, flexible placement, increased velocity of innovation of the data components and control components, but also decouple the data states from the control states, making the states of the data components of the network functions fit into the L1/L2/Last level cache, which is essential for the network functions to scale 100Gbps [25] beyond on COTS (Commercial off-the-shelf) hardware considering the per-packet processing budget is under 6.7 ns.

While the granular decomposition is arguably the right abstraction, realizing it practically is challenging due to: (1) Keeping the cache footprint small is hard when the number of active flows is high. For 100Gbps traffic and beyond, the number of flows is generally high which leads to a large flow table that exceeds the capacity of the cache. (2) The activities of the control/offloading components will negatively impact the performance of the data components. (3) Shared states among the cores will negatively impact the performance.

We observe an opportunity to solve those challenges by (1) a smaller fast table for storing per-flow states. This is driven by the insight that prioritizing flows with the higher occurrence or higher priority for efficient utilization of the private cache of the flow. (2) a runtime system that supports flexible placement of the data components and control/offloading components, placing them on different cores when those cause contention, cache allocation technology to limit the resources usages of actions with lower priority. (3) a mechanism that delays the synchronization of shared states, driven by the insight that shared states do not need updates on a per-packet basis.

We present GranularNF (Section 4), a micro-architecture-aware serverless network platform/abstraction that supports

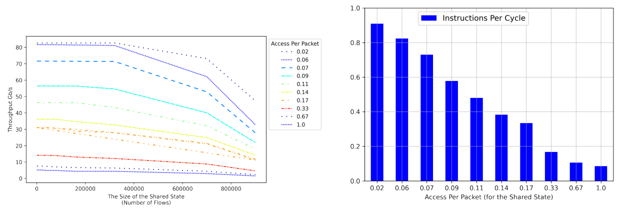


Figure 3: Shared States with High Access Frequency Impacts Performance

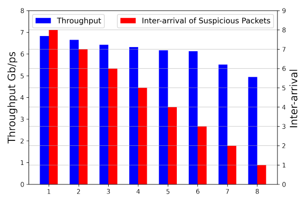


Figure 4: Frequency of Suspicious Packets Impacts Performance of IDS

the aforementioned programming framework on the COTS hardware. GranularNF consists of a director as the control plane and the runtime as the data plane of the system. It provides an interface for the application developers to define data/control/offloading action/states without worrying about the detail of scheduling and the burdensome procedures of upgrading components. We illustrate the benefit (section 5) of the system using a load balancer, IDS, and 5G UPF as examples. We have shown that (1) the granular decomposition can improve the system throughput by better utilization of the cache. (2) the system inherits the benefits of SBA design: horizontally scalable, and velocity.

2 MOTIVATION

We argue that the existing monolithic abstraction of network functions is not sufficient. Scaling stateful NFs imposes serious challenges due to (i) states that cannot fit into the private cache of a core lower the performance (ii) shared states among the NF instances result in the diminishing return in scaling (iii) (high frequency of) calling of routines that have computational complexity impacts the data plane performance. All these three factors will be influenced by the dynamic traffic patterns during the run time. We use three common NFs to illustrate the challenges mentioned.

The first example is a stateful load Balancer, and we measure its single-core performance. NF instances with a high number of flows (large working set) result in bad performance. As illustrated in figure 2, when the number of flows grows, and the state can no longer fit in the cache, the L1 miss rate increases, and the utilization of the L2 and L3 cache significantly degrades. For the optimal placement of NF instances, it is essential to be aware of the size of the working set of the corresponding instance.

To illustrate the performance bottleneck caused by shared states, consider the traffic accounting of a UPF with shared states among network instances. For per-packet network monitoring, shared states with high accessing frequency will impact performance (figure 3). The higher frequency of access to the shared state is, the more performance degradation. The cause is the contention of the access to the data structure as well as the contention in the LLC. For those states evicted from the LLC, the associated operations become memory-bound [26].

Taking IDS as an example, higher network traffic can causes non-negligible computation overhead. In the example, we send evenly balanced traffic to each core but with a different number of suspicious flows. Instances receiving the suspicious traffic will go through a series of calculations (pattern matching) to make decisions on whether or not to blacklist the flows. As illustrated in the figure 4, the overhead is unevenly distributed on each instance even though the throughput of the input traffic is the same.

These examples show how the performance of the NF can be influenced by different characteristics of the components. These behaviors motivate us to break down monolithic NFs to give the right abstractions to describe their behavior. Next, we show how we can describe example stateful NFs using our abstractions.

3 PROGRAMMING EXAMPLES

3.1 Load Balancer

The main function of a load balancer (figure 5 left) is to map and rewrite m (public/virtual) destination IP to n (private/physical) IP addresses (*e.g.,* those of n backend servers). The flow mapper is a data action that will map the packet destination IP address to a new server address. The flow mapper contains a classifier that will differentiate whether a flow is a new flow. If it is an old flow, it will rewrite the header based on the mapping. If it is a new flow, it will send the messages to the server selector which is a control component to select the new destination address according to the policy.

3.2 IDS

For an IDS (figure 5 middle), it needs to inspect the content of the traffic to decide whether to blacklist the flow. The data state consists of a white list of flows and a black list of flows and a list of inspection rules. The white-list of flows allows the incoming flows to be forwarded without further inspections. The black-listed flows are blocked. Besides the matching process, the regular expression matching engines will use the inspection rules to examine the payload of the packets. The policy such as the list of inspection rules can be configured by the remote policy server.

3.3 5G UPF

For a 5G UPF (figure 5 right), it acts as a router between the Access Network and Data Network. The data actions will handle uplink and downlink traffic. It will read the data states to map a part of the 5-tuple to the PDR (Packet Detection rule) which defines how to handle the packets (forwarding, buffering, dropping, accounting). The policy of the rules is defined as installed by the SMF (Session Management Function). The control actions handle the request from the SMF and the control states stores the PFCP (Packet Forwarding Control Protocol) session states. The buffer server contains a memory pool, storing the downlink packets when the user devices are idle. While the forwarding table part of the data states can be duplicated among the cores, the accounting part is shared among the cores.

4 SYSTEM DESIGN

We schematically depict the overall system architecture in Fig. 6. The system has two subsystems: the director and the runtime. They communicate with each other through TCP. The director sends commands to configure/control the runtime, and the runtime is responsible for executing network functions.

Director (Fig 6 left) is the control plane of the system, responsible for the logical representation of NF, resources management of the physical machines underlying the runtime, setting up deploying plan, analyzing metrics/logs collected from the run time to support auto-scaling, placement. NF model is a registry of all the available NFs, with the information on the representation (granular decomposition) of an NF, the properties of actions and states, and the dependencies among the actions. Resource Management Module contains information about how many cores the physical machine has, cache size, and memory size. The goal of the runtime model is to characterize the current performance using metrics such as the throughput of each worker and micro-architecture metrics such as L1/L2/L3 cache hit rate. The PCM module [3] on the runtime will send metrics collected to the director periodically. The orchestration layer is where we implement our agent for orchestration tasks (auto-scaling, auto-placement). This module aims to provide an automated solution for reducing the space of scaling and placement strategies for optimal performance and efficiency.

The goal of the runtime (Fig 6 right) is to provide environments for the NFs to execute. More specifically, it is responsible for fast packet receiving/sending, scheduling different actions to execute and route the messages to them correctly. The worker has exclusive resources (CPU times,

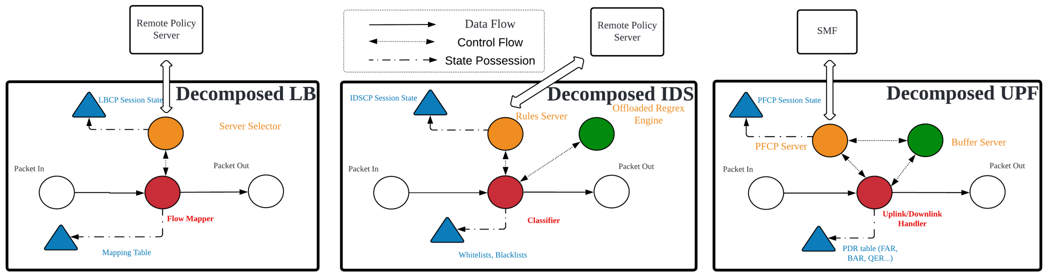


Figure 5: Decomposed NF Examples: (left) load balancer; (middle) IDS: (right) UPF

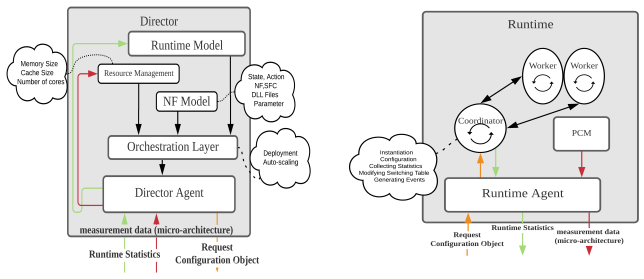


Figure 6: (left) Director; (right) Runtime

cache, NIC queues, RSS buckets) of a core. A worker also has a mailbox implemented by a ring structure to send messages to other workers. When a worker gets the next action, if the core id of the action is on the other core, the worker will put the message into the mailbox and the destination worker can receive it using its mailbox.

Data State Management: It is hard to fit the state into the cache because the number of flows is high hence a larger flow table. But the performance can be improved if we exploit the spatial locality of the flow table and the temporal locality of the arrived flows. For each flow table in the data state, we maintain a fast table and a slow table. The fast table stores the most frequently accessed flows and it is more compact to leverage the spatial locality. The slow table stores all the mapping of the flows. The runtime will periodically update the fast table based on the cache miss rate provided by the profiling modules and the arrival pattern for the flows.

For the shared state management, we make a trade-off between the freshness of the shared states and the performance. Instead of updating the shared state on a per-packet basis, we let each core running data actions keep its own private state. The coordinators will periodically do a merging on the private states of each data action. By sacrificing the freshness of the shared states, we avoid the downside of frequent synchronization of different cores to access the shared states.

Minimizing Interference on the data components: While the control components can negatively impact the performance of the data components, the performance requirement of the control components is much lower compared to the data plane, which means that the performance of the control components will not benefit much from the cache and frequent message exchanging. The overhead of frequent message passing can be reduced through batch processing. The CAT (Cache Allocation Technology) of the Intel CPU feature set can help to limit the number of ways that the control components can use to decrease the interference among the data components and control components. For the offloading components, flexible placement of them to other cores/sockets can relieve the bottleneck on the data plane.

Sharding: By sharding, the state of the network functions is partitioned. To achieve better load balancing among each instance, we set the number of shards greater than the number of cores. For better resource utilization, each shard can be freely migrated from one core in the runtime to the other core by configuring the RSS buckets dynamically.

Implementation: We implement the platform using DPDK for fast packet processing. We implement the load balancer from scratch. For the IDS, we use the code from tiny-regex-c[5] to do the regular expression matching for the payload. For the 5G UPF, we implement a version by extracting the dataplane components of the PDR matching, encapsulation, decapsulation, routing from gtp5g[1], the logic of the buffer server, and PFCP request handlers from free5gc [2]. Compared to the kernel module that the original implementation, the kernel-bypass techniques that we apply will also reduce the data plane and control plane interference by eliminating the cost of context switching of the user space and kernel space.

5 EVALUATION

Experiment setup: To evaluate our platform, we use two machines with Intel(R) Xeon(R) Platinum 8168 CPU which

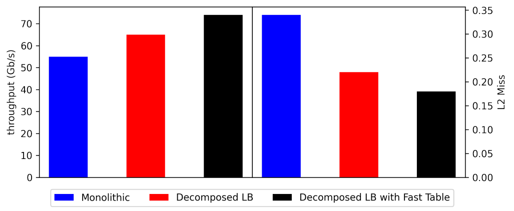


Figure 7: Improvement of the decomposed LB

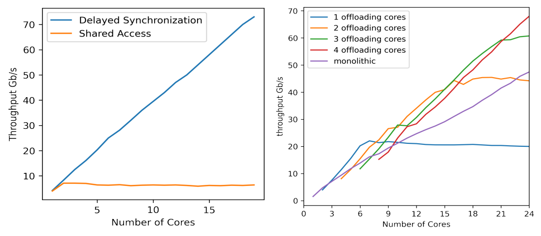


Figure 8: (left) 5G UPF; (right) IDS

has 48 cores on two sockets, and 2 ConnectX-5Ex with 100 Gbps. One machine is running a traffic generator to generate 100 Gbps traffic sent to the second machine which runs our runtime. The director is placed on the traffic generator.

5.1 Data Components Optimization

Improvement of the fast table: We assume that the traffic that is generated contains 50K flows that arrive frequently (90% of the overall traffic) compared to the rest of the flows. For the decomposed network functions, we put the control components on different cores. For all the network functions, we generate simulated policy configuration requests for the control components. Figure 7 compares the performance of the decomposed load balancers to the monolithic ones using 16 cores. The comparison shows the decomposed one is better than the monolithic one because the accesses of the data states and the control states are isolated on different cores. The Fast table version with 5k entries per core can achieve better performance because it consolidates the most frequently accessed flows into a smaller table that can better fit into the cache.

Improvement of the delaying synchronizing: we now evaluate how delaying synchronizing the shared state can improve the performance. We assume that an IP flow is distributed to different cores using RSS. We compare the performance of the 5G UPF with/ without shared states optimization (figure 8). It is shown that with strict per-packet updates of the shared states, the scalability suffers. In comparison, delayed synchronization of the states of different cores can achieve linear scalability.

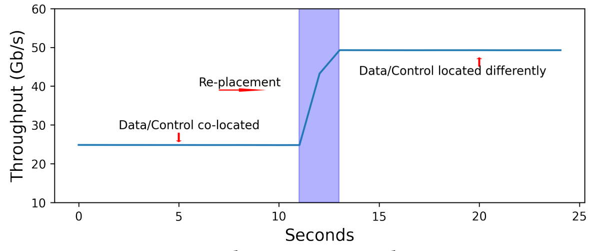


Figure 9: Replacement in the runtime

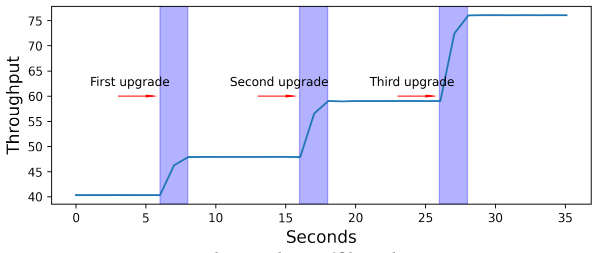


Figure 10: Upgrading the offloading components

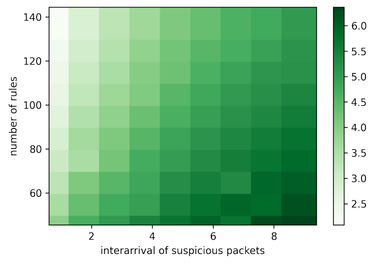
5.2 Velocity

We play a workload that required a deployment scheme that places the control component dynamically on a separate core. Figure 9 shows that our design can support the flexible dynamic placement of each component within a server. More important than the speed of the transition, which is largely attributable to the low-latency nature of the message, is the fact that traffic did not get dropped or need to be steered away from this data plane actor during the transition.

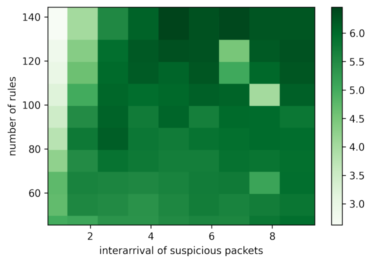
For the second part of the velocity experiment, when running the NF, we periodically upgrade the new offloading component using a dynamic library through the orchestrator. Using the modified NF model in the director, the orchestrator will send instructions to the runtime to replace the new action and change the related data structure accordingly. Figure 10 shows that in our system, the updates can happen without disruption.

5.3 Flexible Placement

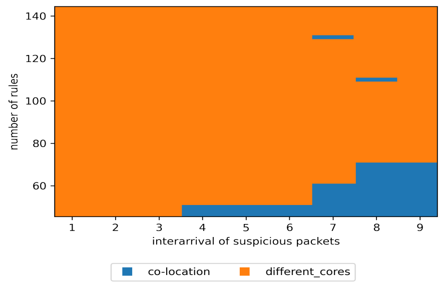
Placement of Offloading actions: First, we evaluate the flexible placement of the regular expression matching engine of the IDS. We generate a workload containing only suspicious flows. We compare the performance of IDS that co-locates the data actions and offloading action on the same core and the one that places them on the different cores. Figure 8 illustrates the relationship between the throughput, the number of cores that run offloading action, and the total number of cores that we use. It shows that when the workload on the regular expression matching engine is high, adding more cores for the data components will not improve the performance. A similar situation happens for using 1,2 and 3 offloading cores. When we use 4 offloading cores, the bottleneck on the data plane is totally removed. The figures



(a) Colocation



(b) Different cores



(c) Comparison

Figure 11: (a) & (b) Performance of network function using colocation/different core placement strategy for offloading components under different frequency and computational overhead. (c) The comparison of the two strategies shows that the optimal placement should depend on a specific workload.

also show that fine-grained resource allocation can lead to efficient resource utilization with less number of cores to achieve the same performance.

Placement Strategy Analysis: Based on different workload characteristics and performance requirements, we have different optimal placement plans – whether we should colocate the data action and offloading action on the same cores. We can adjust the computation overhead of offloading actions of IDS by changing the number of the pattern matching rules it needs to process, and the frequency of its invocations by changing the inter-arrival of suspicious packets. Both figure 11a and figure 11b show that increased computational complexity of offloading actions and decreased inter-arrival of offloading actions lead to performance degradation. By placing them on different cores, figure 11a shows the throughput can be boosted since the work is offloaded to other cores. The comparison in figure 11c shows when each placement plan will outperform the other. When the frequency is lower and the cost of offloading action is higher, placement on different cores tends to perform better. It implies that for optimal performance, we need to change the placement based on the workload in the runtime.

6 RELATED WORK

A number of studies have been devoted to the state management problem for scaling *stateful* NFs and the consistency and robustness implications. For example, Split-&-Merge [20] separates *per-flow* states and *shared* states, with the latter managed centrally; this direction is further extended in [10, 13]. In contrast, Stateless [12] advocates centrally managing all NF states (in a centralized controller), whereas S6[24] employs a DHT key-value store maintained among servers running various NF instances for *distributed* state management. The consistency issues due to *sharing* of NF states are addressed in [14] which advocates using locking mechanisms to ensure correctness; this work is further extended in [11] which employs database techniques. All these studies factor out only the *NF state* from NFs and focus on *scaling across servers*. There are existing serverless NFV platform [19, 21–23], providing a containerized environment for flexible NF deployment, but they do not distinguish *data vs. control* components, which are crucial to scale complex stateful NFs. Without such decoupling and visibility into the NF's stateful operations, it is infeasible to scale existing monolithic "opaque" and "black box" NFs to 100 Gbps and beyond. OpenBox [7] and NetBricks [16] advocate decomposing NFs into reusable modules, but the focus is more on removing redundant operations to speed up packet processing. In contrast, we tackle challenges in scaling stateful NFs by calling for a need to re-architect the NF's state and behavior and develop novel abstractions and a principled approach for refactoring and scaling stateful NFs.

7 CONCLUSION

Without proper design considerations, monolithic scaling solutions with coarse resource allocation fail to scale in the context of 100 Gbps network traffic for modern multi-core architecture. To fill this gap, a finer granularity of visibility by separating the control/data/offloading states and actions of network functions is imperative for optimal performance. We propose a programming framework GranularNF that is useful for the reasoning of scaling network functions. We also provide the runtime and the director, which can support the dynamic scaling of stateful network functions. We empirically show the efficacy of the framework using a load balancer, IDS, and 5G UPF.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their comments. The research was supported in part by NSF under Grants CNS-1814322, CNS-1831140, CNS-1836772, CNS-1901103, CNS-2106771, CNS-2045478 and CCF-2123987.

REFERENCES

[1] [n.d.]. free5gc/gtp5g: GTP-U Linux Kernel Module. https://github.com/free5gc/gtp5g

[2] [n.d.]. free5gc/upf. https://github.com/free5gc/upf

[3] [n.d.]. GitHub - opcm/pcm: Processor Counter Monitor. https://github.com/opcm/pcm

[4] [n.d.]. Home - DPDK. https://www.dpdk.org/

[5] [n.d.]. kokke/tiny-regex-c: Small portable regex in C. https://github.com/kokke/tiny-regex-c

[6] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast userspace packet processing. *ANCS 2015 - 11th 2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (5 2015), 5–16. https://doi.org/10.1109/ANCS.2015.7110116

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

[8] Shihabur Rahman Chowdhury, Haibo Bian, Tim Bai, and Raouf R Boutaba David Cheriton. [n.d.]. μNF: A Disaggregated Packet Processing Architecture. ([n. d.]).

[9] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*. 523–535. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud

[10] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. ACM, New York, NY, USA, 163–174. https://doi.org/10.1145/2619239.2626313

[11] Milad Ghaznavi, Elaheh Jalalpour, Bernard Wong, Raouf Boutaba, and Ali José Mashtizadeh. 2020. Fault Tolerant Service Function Chaining. In *Proc. of ACM SIGCOMM'20* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 198–210. https://doi.org/10.1145/3387514.3405863

[12] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 97–112. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kablan

[13] Junaid Khalid et al. 2016. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr. In *13th USENIX NSDI*. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/khalid

[14] Junaid Khalid and Aditya Akella. 2019. Correctness and Performance for Stateful Chained Network Functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/nsdi19/presentation/khalid

[15] Zili Meng, Jun Bi, Senior Member, Haiping Wang, Chen Sun, and Hongxin Hu. 2019. MicroNF: An Efficient Framework for Enabling Modularized Service Chains in NFV. *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS* 37 (2019). Issue 8. https://doi.org/10.1109/JSAC.2019.2927069

[16] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 203–216. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda

[17] Imtiaz Parvez, Ali Rahmati, Ismail Guvenc, Arif I. Sarwat, and Huaiyu Dai. 2018. A survey on low latency towards 5G: RAN, core network and caching solutions. *IEEE Communications Surveys and Tutorials* 20 (10 2018), 3098–3130. Issue 4. https://doi.org/10.1109/COMST.2018.2841349

[18] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert G. Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. 2013. Ananta: cloud scale load balancing. In *ACM SIGCOMM 2013 Conference, SIGCOMM'13, Hong Kong, China, August 12-16, 2013*. 207–218. https://doi.org/10.1145/2486001.2486026

[19] Matteo Pozza, Ashwin Rao, Diego F Lugones, and Sasu Tarkoma. 2021. FlexState: Flexible State Management of Network Functions. *IEEE Access* 9 (2021), 46837–46850.

[20] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. NSDI*.

[21] Junxian Shen, Heng Yu, Zhilong Zheng, Chen Sun, Mingwei Xu, and Jilong Wang. 2020. Serpens: A high-performance serverless platform for nfv. In *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*. IEEE, 1–10.

[22] Arjun Singhvi, Junaid Khalid, Aditya Akella, and Sujata Banerjee. 2020. Snf: Serverless network functions. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 296–310.

[23] Jianfeng Wang, Tamás Lévai, Zhuojin Li, Marcos AM Vieira, Ramesh Govindan, and Barath Raghavan. 2021. Galleon: Reshaping the Square Peg of NFV. *arXiv preprint arXiv:2101.06466* (2021).

[24] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic scaling of stateful network functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 299–312.

[25] Peng Zheng, Wendi Feng, Arvind Narayanan, and Zhi-Li Zhang. 2020. NFV Performance Profiling on Multi-core Servers. (2020).

[26] Peng Zheng, Arvind Narayanan, and Zhi-Li Zhang. 2019. Towards a Scalable, Flexible and High Performance NFV Execution Model. In *Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies*. 68–69.