Evaluating Network Stacks for the Virtualized Mobile Packet Core

Ashwin Kumar*, Priyanka Naik+, Sahil Patki*, Pranav Chaudhary*, Mythili Vutukuru*
Department of Computer Science and Engineering, Indian Institute of Technology, Bombay*
India
IBM Research+
India
ashkumar@cse.iitb.ac.in,priyanka.naik@ibm.com,{sahilpatki,pranavchaudhary,mythili}@cse.iitb.ac.in

ABSTRACT

Several novel userspace network stacks have been proposed in recent research to overcome the limitations of the Linux network stack in providing high-performance I/O for Virtual Network Functions (VNFs). In this paper, we evaluate the performance of several state-of-the-art network stacks in the context of the VNFs of the 5G mobile packet core. The VNFs in the 5G core are several times more compute-intensive than the VNFs used to benchmark network stacks in prior work, given the need to perform user authentication and other such cryptographic operations. Our evaluation shows that while modern stacks outperform the Linux kernel stack over I/O intensive VNFs (as observed in prior work), the performance gap is not as wide in the case of CPU-intensive VNFs of the 5G core. We also find that the packet core VNFs can obtain up to 67% higher performance if the network stack could partition traffic to CPU cores at the granularity at which VNFs maintain state (mobile subscriber in this case), enabling a lockfree architecture within the VNF. The insights from our work can help us design a network stack that is better suited for compute-intensive VNFs such as those in the 5G core.

CCS CONCEPTS

• **Networks → Network performance analysis**; *Network simulations*; *Mobile networks.*

APNet 2021, June 24–25, 2021, Shenzhen, China, China
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8587-9/21/06…$15.00
https://doi.org/10.1145/3469393.3469402

KEYWORDS

Network Function Virtualization, 5G core, Cellular networks, Kernel network stack, Kernel bypass, DPDK

ACM Reference Format:
Ashwin Kumar*, Priyanka Naik+, Sahil Patki*, Pranav Chaudhary*, Mythili Vutukuru*. 2021. Evaluating Network Stacks for the Virtualized Mobile Packet Core. In *5th Asia-Pacific Workshop on Networking (APNet 2021) (APNet 2021), June 24–25, 2021, Shenzhen, China, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3469393.3469402

1 INTRODUCTION

Network Function Virtualization (NFV) is a paradigm of building packet processing network functions as software entities running on commodity hardware, instead of as custom packet processing hardware. NFV promises to provide several benefits like lower cost of developing software (as compared to longer hardware development cycles) and the ability to scale to higher loads by scaling "horizontally" via spawning new software replicas (as opposed to scaling "vertically" by upgrading to more powerful hardware). Several Virtual Network Functions (VNFs) are widely available for enterprise networks today [7, 19, 20, 22, 40], to be deployed on baremetal or within virtual machines and containers within the cloud. The most interest for NFV has come from telecommunication networks, with network functions in the mobile packet core being standardized as VNFs in the upcoming 5G standards. The mobile packet core connects the radio access network (comprising mobile subscribers and base stations) to external networks. The control plane of the packet core consists of network functions that perform subscriber registration, authentication, session management, and other signaling procedures, while the data plane network functions forward user traffic. With high signaling traffic expected from IoT and other new applications [2, 13, 24, 28, 45], a mobile packet core with high control plane throughput and low latency is imperative for the success of 5G. While the previous generations of telecommunication networks built these network functions as custom hardware boxes, the softwarization of these network functions began in 4G (LTE) networks and has become the de facto way of architecting 5G networks [2].

Previous research has documented the limitations of the Linux kernel network stack in handling high-speed I/O that is required for NFV [46]. While there have been attempts to fix the packet processing subsystem in the Linux kernel over the years [25, 36, 41, 53], the more popular option has been to use kernel-bypass techniques like DPDK (Data Plane Development Kit [29]) and netmap [46] to process packets directly in userspace applications with low overhead. Some VNFs can just work off the raw packets delivered by the kernel bypass mechanisms, but other VNFs terminate transport layer endpoints (e.g., HTTP proxy server) and must perform TCP/IP network stack processing. Therefore, several userspace network stacks have been proposed to enable TCP/IP processing in userspace for applications that perform I/O using kernel bypass mechanisms [10, 12, 27, 31, 33, 34, 37, 39, 43]. Kernel bypass techniques, in combination with optimized userspace network stacks, have enabled VNFs to process network packets at linerate on high-speed network links, and have ameliorated the concern of whether software network functions can match the performance of their hardware counterparts.

In this paper, we investigate the question of which network stacks are suitable to deploy alongside the VNFs in the control plane of the mobile packet core. The question merits investigation because previous work on comparing the performance of various kernel and kernel bypass stacks has focused mostly on I/O-intensive network functions like load balancers, firewalls, or RPC servers. On the other hand, the control plane VNFs in the mobile packet core perform CPU-intensive tasks such as user authentication, and it is not clear if the conclusions arrived earlier hold for this very different workload as well. We used several standards-compliant network functions of the 5G packet core and evaluated their performance across several state-of-the-art network stacks, for a workload involving registering and authenticating mobile subscribers in the packet core. We found that the performance gap between the Linux network stack and the various userspace stacks was much narrower than that reported in prior work—the kernel stack has only 23% lower throughput than the best performing userspace stack, and even performed on par with some of them. We verified in our setup that kernel bypass stacks continued to outperform the Linux network stack by a factor of 5 or more on the more traditional I/O intensive workloads. This conclusion (which is somewhat obvious, in hindsight) can be explained by the fact that the network stack processing takes up less than 25% of CPU cycles in case of CPU-intensive VNFs, a much lower share than with I/O-intensive ones. As a result, any overheads in the processing of the kernel network stack have a much smaller performance impact, especially in the case when the connections between the compute-intensive VNFs were persistent (causing lower connection setup/teardown processing in the stack). Furthermore, as we elaborate later (§4), some design decisions in network stacks perform differently for different VNF workloads. These results lead us to the conclusion that there is no one network stack that works for all VNFs, and the design of the optimal network stack is closely tied to the characteristics of the VNF packet processing. Existing research that uses a homogenous set of benchmarks to compare network stack performance will benefit from widening its ambit to include diverse VNFs such as those of the mobile packet core.

Experimenting with the VNFs in the mobile packet core also led us to realize another point of difference between these VNFs and those traditionally used for network stack performance evaluation. Most network functions used in prior work (e.g., NATs) naturally maintain application state at TCP-flow granularity, while the VNFs in the packet core maintain state at the granularity of mobile subscribers. This distinction is important because userspace network stacks scale their performance with increasing load by distributing incoming traffic to CPU cores using the hash of the TCP 4-tuple [26], ensuring that traffic of a TCP connection is processed on the same core throughout the lifetime of a connection [6, 12, 27, 31, 36, 47]. This partitioning of traffic means that the VNFs built on such multicore scalable stacks can also maintain their TCP flow state in per-core local data structures, and eschew locking across CPU cores for the most part. Such optimizations are not possible in the mobile packet core VNFs because the traffic of a mobile subscriber can arrive on any core, requiring locking to access all application state in a multicore VNF. Our preliminary experiments show that if it were possible to isolate the traffic of a mobile subscriber to a single CPU core within the network stack, then the VNF could improve its performance by up to 67%, by eliminating locking overheads when running across multiple CPU cores. Of course, it is very challenging to partition traffic based on mobile subscriber identity within the network stack, as this information is not available in the traditional TCP/IP headers parsed by the network stack. A network stack design that can surmount this challenge can significantly improve performance for the VNFs of the mobile packet core—a requirement that future research in this space should keep in mind. The ability to parse complex headers in programmable hardware [5, 11, 32, 48] can be explored as a possible way to perform VNF-aware packet steering [18] in future stacks.

2 BACKGROUND AND RELATED WORK

VNF Taxonomy. VNFs are software appliances that process network packets destined to end hosts in a network. While some VNFs manipulate packet headers and operate at the network layer (e.g., routers, firewalls, NATs), other

VNFs act as transport layer clients/servers and operate at the application layer of the TCP/IP stack. A VNF performs some processing on receiving a packet, and we can classify a VNF as compute (CPU) intensive or I/O intensive based on the amount of processing that needs to be done for each received packet. Further, for VNFs that terminate transport layer endpoints, the VNFs can open a new connection for each request that must be processed (short connections) or send multiple requests over the same connection (persistent connections). Table 1 categorizes VNFs used in prior work along these dimensions. We see from the table that there are several VNFs that fall under the CPU-intensive category.

Mobile packet core. A mobile network has two parts: the radio access network (RAN) consists of the User Equipment (UE) and the base station, and the packet core connects the RAN to external networks. Figure 1 shows the architecture of a 5G network. The control plane of the packet core consists of the Access and Mobility Function (AMF), Session Management Function (SMF), Authentication Service Function (AUSF) and other components that together manage user registration, authentication, session management, handovers, and other signaling procedures. The data plane consists of the User Plane Function (UPF) that routes and forwards encapsulated user traffic. The control plane components communicate with each other over REST-based HTTP interfaces, and the AMF communicates with the base station over SCTP. Therefore, the VNFs in the control plane terminate transport layer endpoints and must run over a TCP/IP network stack.

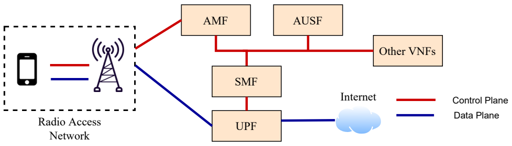

Figure 1: 5G core architecture.

Among the multiple signaling procedures handled by the control plane of the mobile packet core, we focus on the registration procedure in this paper. When a mobile subscriber wishes to begin a data communication, she must register and authenticate herself with the mobile network. The user's initial registration request is sent via the base station to the AMF, which in turn communicates with the AUSF and other VNFs to complete the user authentication. Processing of the registration request involves at least two rounds of request/response communication between the AMF and AUSF. Upon receiving a registration request, AMF first obtains the authentication challenge from the AUSF, relays it to the user, and conveys the user's response back to AUSF again for verification. While we omit a detailed discussion of the registration callflow, it suffices to know that the processing involves cryptographic computations and is reasonably compute-intensive.

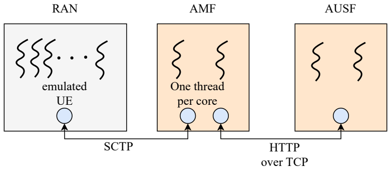

Figure 2: 5G Core VNF Design.

Network stacks for NFV. VNFs that terminate transport layer endpoints must work alongside a network stack that performs TCP/IP processing. VNFs built over the socket interface of the kernel use the kernel's TCP/IP stack by default, but the kernel's mechanisms of system calls, interrupts, locking overhead across shared kernel data structures, and dynamic allocation of packet memory, among other things, have been found to be an impediment towards achieving high throughput I/O for such VNFs [46]. While some attempts have been made to fix the bottlenecks in the Linux network stack [25, 36, 41, 53], other attempts have focused on bypassing the kernel altogether and processing packets directly in userspace via packet I/O mechanisms like DPDK [29] and netmap [46]. VNFs using such kernel bypass techniques will receive raw packets, and will have to use a userspace network stack to process the received packets. One of the earliest such stacks, mTCP [31], incorporates optimizations like per-core local TCP/IP data structures and batching to achieve multicore scalable network stack processing in userspace. While the mTCP processing runs in a separate thread that is colocated with the application thread on each CPU core, TAS (TCP acceleration as a service) [34] proposes running network stack processing on dedicated cores, separate from the application. TAS assigns separate cores to handle the TCP connection setup and teardown on a slow path, and runs the TCP data handling on the fast path. Arrakis [42] and IX [10] seek to eliminate kernel overheads on the datapath of the network stack, and use the virtualization support in network cards to give each application its own slice of the NIC for optimized processing. IX uses a run-to-completion model to perform network stack and application processing in the same thread on a CPU core. ZygOS [43] and Shinjuku [33] extend the idea of IX to workloads which have variable-sized requests, and use mechanisms like work-stealing and preemption to balance load across the multiple cores on which the network stack runs, thereby reducing tail latencies of applications. Shenango [39] focuses on I/O intensive workloads and improves CPU efficiency by reducing busy spinning in the network stack via reallocation of CPU across applications at a fine granularity. A survey of the benchmarks used in the evaluation of these stacks indicate that most of them focus on the I/O intensive workloads of Table 1. Our work focuses on evaluating a subset of these network stacks on the CPU-intensive VNFs of the 5G core.

	Persistent Connection	Short Connection
CPU Intensive	Suricata [22], endRE (WAN optimizer) [4], Ngnix [20], Squid [49], BRAS [15], mobile packet core control plane [3]	Bro [40], Zeek [44], Kargus [30]
I/O Intensive	twemproxy [50], Mcrouter [21], Apache traffic server [7], Netflix-ribbon [38], Abacus [23], mobile packet core data plane [1]	PRADS [51], HAProxy [14], Varnish Cache [52], Faild [8]

Table 1: Taxonomy of VNFs

3 5G CORE VNF DESIGN

In this section, we describe the architecture and implementation of the 5G core VNFs that are used to benchmark various network stacks (Figure 2). Our implementation builds upon the standards-compliant 5G VNFs obtained from [16, 17]. We focus our attention on AMF and AUSF, which are involved in handling a UE's registration request, and build several variants of these for our experiments. We build emulators for the other VNFs that are not part of our study.

Load generator. We use a RAN emulator to pump traffic to the 5G core VNFs. Our RAN emulates the behavior of a configurable set of K concurrent UEs in a closed loop manner, with K being varied to vary the load in our experiments. We only emulate registration requests from the UEs; considering a wider set of signaling procedures is part of future work. The RAN communicates with AMF over an SCTP connection, on which the UE registration requests are sent, after being encapsulated in various 5G-specific protocol headers. We use more than one RAN to saturate AMF capacity in our experiments; in such cases, each RAN uses a separate SCTP connection to the AMF, much like in real-life.

AMF. AMF is a multi-threaded VNF with N threads, with each thread pinned to a CPU core and spinning in an event-driven epoll loop. When measuring the performance of AUSF, we chose N to be large enough for AMF to generate enough load to saturate AUSF. UE requests from the RAN arriving over SCTP are distributed to the various AMF cores using RSS. Handling a registration request requires the AMF to send two different HTTP requests to AUSF. Upon first receiving the registration request, AMF requests and obtains an authentication challenge from AUSF. This authentication challenge is sent to the UE via the SCTP connection, and a response from the UE is obtained. This UE response is once again sent to AUSF by AMF to have it verified. Both these HTTP transactions happen over separate TCP connections between AMF and AUSF.

AMF-Persistent. In order to generate a workload with a lower overhead of TCP connection setup/teardown, we build a persistent connection variant of AMF that uses a set of long-lived TCP connections to AUSF and reuses the same connection to send multiple authentication requests.

AMF-Lockless. To scale network processing across cores, RSS splits traffic to cores using the hash of the TCP 4-tuple, ensuring that traffic of a TCP connection goes to the same core always. As a result, VNFs that maintain state at TCP flow granularity can use a lockfree multicore scalable architecture by storing state within per-core local datastructures. However, AMF and other packet core VNFs maintain state at the granularity of a mobile subscriber. For example, the AMF maintains per-UE registration context for the lifetime of a UE's registration with the network. This implies that the requests of a UE can be processed at any of the cores of a VNF, requiring the VNF threads to access UE state using a lock for mutual exclusion. In order to understand the potential performance gains of a lockfree AMF design, we build a lockfree variant of AMF, where each of the N AMF threads store UE context in per-core local data structures. To ensure that traffic of a UE arrives at a single core only, we use N different RAN emulators sending traffic on separate connections for a disjoint set of UEs, so that the N AMF threads, each talking to a separate RAN, can safely update UE context without locking. Note that this is not a very practical design for real-life AMFs, unless the underlying network stack is somehow made aware of UE identifiers and splits traffic to cores based on these identifiers. However, we use this AMF variant to help us understand if building such awareness into network stacks is worth the effort in the first place.

AUSF. AUSF listens for AMF requests on a HTTP server socket in an event-driven epoll loop and responds to them suitably. Note that each AMF request requires significant CPU computation at the AUSF, measured to be around 300K CPU cycles in our implementation, which is several orders of magnitude higher than that considered in prior work. We use multiple event-driven threads to scale AUSF to multiple cores as required. In addition to an implementation over the kernel epoll API, we also build variants of AUSF that run over the mTCP, TAS and IX network stacks. These network stacks were chosen because they span a wide spectrum of design options in userspace stacks. AUSF over IX executes its request processing in a run-to-completion model. With mTCP, the AUSF processing runs in a separate thread that is co-located with the mTCP thread on the same core. With TAS, AUSF processing runs on a separate core from the network stack processing of TAS. We do not consider other network stacks like Shinjuku and Shenango presently because they are optimized for a heterogeneous request processing workload while our AUSF currently processes only one type of request.

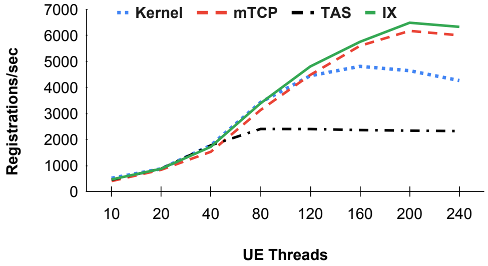

Figure 3: AUSF: Network stack throughput

AUSF-IO. In order to reproduce the results of prior work, we also built an I/O intensive variant of AUSF, which does no computation on received requests and echoes back a dummy response. We modify our load generator suitably for such requests and responses. We built variants of this I/O intensive AUSF on the same set of network stacks: the Linux kernel stack, mTCP, IX, and TAS.

4 EVALUATION

For our evaluation, we use two Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz servers with 24 cores and 128 GB memory each, connected through Intel 82599ES 10GbE NICs. The RAN emulator and AMF run on one server over the Linux network stack, while the AUSF runs on the other server over different network stacks across experiments. We use the Ubuntu 18.04 with kernel 5.0.0-37 for experiments with the Linux kernel stack, mTCP and TAS, and Ubuntu 14.04 with kernel 4.4.0-142 for IX. For all experiments, we ensure that all VNFs except the one under test have enough CPU and RAM to not become the performance bottleneck. We generate enough load from the load generator to saturate the VNF under test, and measure the throughput (registration requests processed/sec) and end-to-end latency (completion time of the registration procedure) averaged over 300 seconds.

4.1 Network stack comparison

Figure 3 shows the throughput (in terms of number of registration requests processed/sec; we saw no failures) of the compute-intensive AUSF running on a single core, when using different network stacks, as a function of offered load (number of concurrent UEs sending registration traffic). Increasing the offered load beyond 240 concurrent UEs did not increase performance due to CPU saturation at AUSF. We see from the figure that IX performs the best, followed by mTCP, the Linux kernel stack, and TAS. We note that the performance of the Linux network stack is not far off from the more advanced userspace stacks—its throughput is only 23% lower than that of IX at saturation. The latency measurements also show a similar trend. These results show that *the Linux kernel stack is not a bad alternative for compute-intensive VNFs like those of the mobile packet core*, especially when kernel bypass stacks are harder to setup in shared cloud environments with little control over the networking subsystem. We also repeated this experiment with AUSF running on 2 CPU cores, and verified that AUSF throughput almost doubled, confirming that AUSF has a CPU bottleneck.

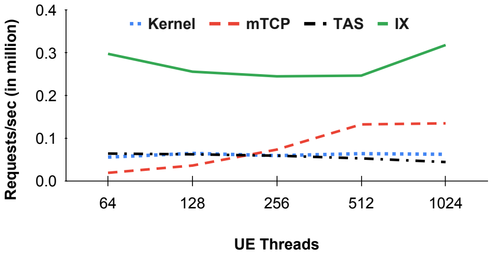

Figure 4: AUSF-IO: Network stack throughput.

We use the profiling tool perf [35] to understand the reasons behind these performance results. Across all stacks, we find that the network stack processing takes less than 25% of CPU cycles, while AUSF consumes most of the CPU. Therefore, any performance gains from the usage of complex network stacks are bound to have a limited impact. Our profiling also explains the somewhat poor performance of TAS. We find that the userspace networking library used by TAS applications (libtas) consumes a lot of CPU cycles for connection context management due to the frequent connection setup and teardown in our workload, starving the VNF of CPU cycles to perform its computations. We also observe that while IX performs well in this experiment, the optimized IX userspace library only supports networking functionality and not other types of I/O. IX uses virtualization support for giving applications a dedicated path to the NIC. Therefore, IX can suffer from poor performance when the application performs system calls for other types of I/O, as well as when applications face an increased number of page faults, as both of these will need to be handled in the kernel via a VM exit [9, 10].

Next, we evaluate the network stacks on an I/O intensive workload, to reproduce the trends from prior work. Figure 4 shows the throughput of all network stacks with the I/O-intensive variant of AUSF. We find that the kernel bypass stacks significantly outperform the Linux kernel stack as reported in prior work, and IX has 5 times higher throughput than the kernel stack. Note that with IX, we exhausted CPU cores at our load generator before IX hit saturation, so our results for the best performing userspace stack are only a lower bound. From our profiling results, we find that system call overhead and other factors slow down the kernel stack as expected. We find that the performance of TAS is limited by the overhead of frequent connection setup and teardown, which are handled on the slow path. The performance of mTCP is lower than that of IX because IX runs

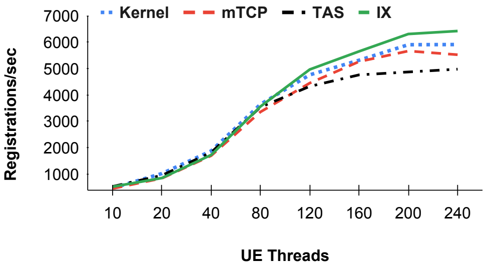

Figure 5: AUSF: Throughput with AMF-Persistent

both VNF and network stack processing within the same thread in a run-to-completion manner, while the network stack processing in mTCP runs in a separate thread that is co-located on the same core as the VNF thread. This two threaded model introduces context switch overheads which lowers performance of mTCP slightly as compared to IX.

Next, we use the persistent connection variant of AMF to communicate with the regular CPU-intensive AUSF, and compare the throughput of AUSF across all network stacks. The results are shown in Figure 5. We find that the performance of the Linux network stack and TAS have significantly improved in this case as compared to the previous experiments. TCP connection setup and teardown requires access to shared kernel datastructures in the kernel stack, and is handled on the slowpath in TAS, so the use of persistent connections eliminates this overhead and significantly improves the performance of both these stacks. We find that the performance of mTCP falls slightly due to the complex interaction between the mTCP and VNF threads once again—the mTCP thread is very lightly loaded in the case of persistent connections, and sometimes sleeps for long durations of time, depriving the application thread of enough work.

In summary, the question of which network stack is best is closely tied to the specific VNF workload, and there is no one network stack that works for all cases. Network stacks designed with one set of workloads in mind can perform very differently when subjected to a different type of workload. Our work shows that network stack benchmarks should expand beyond the traditional I/O intensive VNFs and RPC servers, to more real-world VNFs with complex characteristics. The network stack should also be designed in a way that is robust across changes in VNF workload characteristics. As part of future work, we plan to use more 5G core VNFs and a wider set of signaling messages, and test the various network stacks for these diverse set of workloads. We also plan to explore stacks like Shinjuku [33] when we move to testing with signaling procedures with varying processing times.

4.2 Lockfree VNF design

Next, we evaluate the impact of building application awareness into the network stack. So far, our experiments have used the regular AMF that stores per-UE context in a global datastructure, which all AMF threads access with locking. We also designed a lockless variant of AMF as described in §3, by simulating a network stack that partitioned network traffic to CPU cores at the UE granularity, allowing AMF to store UE context in per-core local datastructures without locking. We provide enough CPU cores to AUSF to ensure that it is not a bottleneck in this experiment. Figure 6 shows the performance of the regular and lockless variants of AMF as a function of number of CPU cores. We find that the lockfree AMF significantly outperforms the regular locking-based AMF, achieving 67% higher throughput at 24 cores. This result shows that incorporating an awareness of the granularity at which VNFs store state into the network stack, though challenging at first sight, might be a worthwhile direction to pursue in future research.

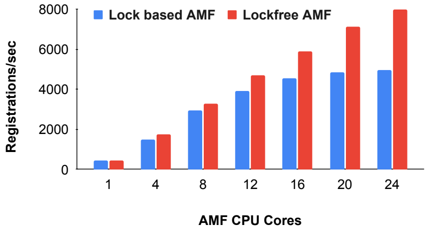

Figure 6: Comparison of lock-based and lock-free AMF

5 CONCLUSION

This paper compares the performance of the state-of-the-art network stacks for CPU-intensive VNFs of the 5G mobile packet core, a design point often ignored by prior work. Our results show that the performance of a network stack varies significantly based on the VNF workload. For example, the kernel network stack, which was shown to perform much worse than newer userspace stacks in prior work, performs relatively better with CPU-intensive VNFs. This paper presents several such differences in the performance of network stacks in the context of the 5G core VNFs. Further, we also show that a network stack that is aware of the granularity at which the application maintains state, and partitions traffic to CPU cores at this granularity, can lead to significant improvements in VNF performance. While incorporating VNF semantics into the lower level network stack can seem challenging at first, newer advances like programmable dataplane hardware with programmable packet parsers can be employed to surmount this technical challenge. The insights of our work can serve as a starting point towards improving the network stack design for VNFs of the mobile packet core.

ACKNOWLEDGMENTS

We would like to thank the 5G testbed project funded by the Department of Telecommunication (DoT), Government of India, for access to the 5G packet core components.

REFERENCES

[1] 3GPP. 2018. The Evolved Packet Core. http://www.3gpp.org/technologies/keywords-acronyms/100-the-evolved-packet-core.

[2] 3GPP. 2020. 3GPP Specification Set: 5G. https://www.3gpp.org/dynareport/SpecList.htm?release=Rel-15&tech=4.

[3] 3GPP. 2020. Access and Mobility management Function (AMF). https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3445.

[4] Bhavish Aggarwal, Aditya Akella, Ashok Anand, Athula Balachandran, Pushkar Chitnis, Chitra Muthukrishnan, Ramachandran Ramjee, and George Varghese. 2010. EndRE: An End-System Redundancy Elimination Service for Enterprises. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation.*

[5] Ashkan Aghdai et al. 2018. Transparent Edge Gateway for Mobile Networks. In *IEEE 26th International Conference on Network Protocols (ICNP).*

[6] Abdul Alim, Richard G. Clegg, Luo Mai, Lukas Rupprecht, Eric Seckler, Paolo Costa, Peter Pietzuch, Alexander L. Wolf, Nik Sultana, Jon Crowcroft, Anil Madhavapeddy, Andrew W. Moore, Richard Mortier, Masoud Koleni, Luis Oviedo, Matteo Migliavacca, and Derek McAuley. 2016. FLICK: Developing and Running Application-Specific Network Services. In *2016 USENIX Annual Technical Conference (USENIX ATC 16).*

[7] Apache. 2020. Traffic Server. https://github.com/apache/trafficserver.

[8] João Taveira Araújo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. 2018. Balancing on the Edge: Transport Affinity without Network State. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation.*

[9] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12).*

[10] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14).*

[11] Carmelo Cascone and Uyen Chau. 2018. Offloading VNFs to programmable switches using P4. In *ONS North America.*

[12] Tencent Cloud. 2019. f-stack. http://www.f-stack.org/.

[13] FCC Technological Advisory Council. 2018. 5G Edge Computing Whitepaper. https://transition.fcc.gov/bureaus/oet/tac/tacdocs/reports/2018/5G-Edge-Computing-Whitepaper-v6-Final.pdf.

[14] Haprxy Developers. 2019. HAProxy. http://www.haproxy.org/.

[15] T. Dietz, R. Bifulco, F. Manco, J. Martins, H. Kolbe, and F. Huici. 2015. Enhancing the BRAS through virtualization. In *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft).*

[16] DoT. 2020. 5G Testbed @ CSE IITB. https://www.cse.iitb.ac.in/~5gtestbed/.

[17] DoT. 2021. 5G Testbed. https://5gtestbed.in/.

[18] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. 2019. Partition-Aware Packet Steering Using XDP and EBPF for Improving Application-Level Parallelism. In *Proceedings of the 1st ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms.*

[19] ETSI. 2020. Network Functions Virtualisation (NFV). https://www.etsi.org/technologies/nfv.

[20] F5. 2020. Ngnix. https://www.nginx.com/.

[21] Facebook. 2020. Mcrouter. https://github.com/facebook/mcrouter.

[22] Open Information Security Foundation. 2020. Suricata. https://suricata-ids.org/.

[23] Younghwan Go, Jongil Won, Denis Foo Kune, EunYoung Jeong, Yongdae Kim, and KyoungSoo Park. 2014. Gaining Control of Cellular Traffic Accounting by Spurious TCP Retransmission. In *Proc. of Network and Distributed System Security Symposium (NDSS)'14.*

[24] GSMA. 2019. Internet of Things in the 5G Era: Opportunities and Benefits for Enterprises and Consumers. https://www.gsma.com/iot/wp-content/uploads/2019/11/201911-GSMA-IoT-Report-IoT-in-the-5G-Era.pdf.

[25] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation.*

[26] Tom Herbert and Willem de Bruijn. 2018. Receive Side Scaling (RSS). https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[27] Michio Honda, Felipe Huici, Costin Raiciu, Joao Araujo, and Luigi Rizzo. 2014. Rekindling Network Protocol Innovation with User-level Stacks. *SIGCOMM Comput. Commun. Rev.* (2014).

[28] IBM. 2017. 5G Will Accelerate a New Wave of IoT Applications. https://newsroom.ibm.com/5G-accelerate-IOT.

[29] Intel. 2018. Intel Data Plane Development Kit. http://dpdk.org/.

[30] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. 2012. Kargus: A Highly-Scalable Software-Based Intrusion Detection System. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security.*

[31] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. MTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation.*

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

[33] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for µSecond-Scale Tail Latency. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation.*

[34] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019.*

[35] Kernel.org. 2020. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[36] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. 2016. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems.*

[37] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles.*

[38] Netflix. 2020. Ribbon. https://github.com/Netflix/ribbon.

[39] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation.*

[40] Vern Paxson. 1999. Bro: A System for Detecting Network Intruders in Real-Time. *Comput. Netw.* (1999).

[41] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. 2012. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*.

[42] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*.

[43] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*.

[44] The Zeek Project. 2020. An Open Source Network Security Monitoring Tool. https://zeek.org/.

[45] Qualcomm. 2020. What is 5G. https://www.qualcomm.com/invention/5g/what-is-5g.

[46] Luigi Rizzo. 2012. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*.

[47] ScyllaDB. 2019. Seastar. http://seastar.io/.

[48] Rinku Shah, Vikas Kumar, Mythili Vutukuru, and Purushottam Kulkarni. 2020. TurboEPC: Leveraging Dataplane Programmability to Accelerate the Mobile Packet Core. In *Proceedings of the Symposium on SDN Research*.

[49] Squid. 2020. Squid: Optimising Web Delivery. http://www.squid-cache.org/.

[50] Twitter. 2019. Memcached Proxy. https://github.com/twitter/twemproxy.

[51] Ubuntu. 2020. PRADS. http://manpages.ubuntu.com/manpages/eoan/en/man1/prads.1.html.

[52] Varnish-Cache. 2020. Varnish Cache. http://varnish-cache.org/.

[53] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*.