Containerized Design and Realization of Network Functions Virtualization for a Light-Weight Evolved Packet Core Using OpenAirInterface

Wen-Ping Lai*, Yong-Hsiang Wang, Kuan-Chun Chiu
Department of Electrical Engineering
Yuan Ze University
Taoyuan, Taiwan
wpl@saturn.yzu.edu.tw*

***Abstract*—In recent years, *network functions virtualization* (NFV) has been well perceived as the driving force behind innovations of the 5G system, such as slicing precious system resources for differential service needs. In this paper, we propose a *container-based* design of *virtual evolved packet core* (vEPC) *slice* and its *light-weight* version (LW-vEPC) based on the OpenAirInterface (OAI) software package. We have successfully containerized, and thus virtualized, the EPC component functions into two separate containers: the *control-plane* (CP) container for *virtual home subscriber server* (vHSS) and *virtual mobility management entity* (vMME), and the *data-plane* (DP) container for *virtual serving and packet data network gateway* (vSPGW). Via a joint configuration design of *virtual linking*, *binding* and *bridging*, including appropriate *source* and *destination network address translation* (SNAT and DNAT), both the *intra*-container and *inter-container* communications have been successfully realized. An OAI-based joint test of vEPC with a small-cell base station (ENB) has also been successfully demonstrated via a downlink video streaming showcase from the Internet to a cellular phone. The DP container itself can also perform as a LW-EPC slice near the mobile edge of ENB to greatly reduce the latency for time-critical applications. The resource allocation methodology of multiple CPU cores for vEPC and LW-EPC slicing is being developed. This paper proposes a simple but powerful algorithm called *specifically assigned cores* (SAC) to achieve better utilization of CPU cores. Our preliminary results show that SAC outperforms the default scheme, namely *randomly assigned cores* (RAC), in terms of *lower CPU load* and *less packet loss*. The superiority of SAC over RAC amplifies with the traffic level.**

Keywords- 5G; NFV; SDN, Cloud, SAC, network slicing; container; Docker

I. INTRODUCTION

The coming era of the *fifth generation* (5G) [1] mobile communication and networking is not only remarked with its much wider communication bandwidth by introducing the use of 5G *new radio* (5G-NR) at high-frequency mm-wave bands, but more with its networking innovations by embracing new ideas from the datacom making itself rapidly evolve from closure to openness. The 5G networking innovations are driven by the maturity and wide deployment of Cloud-based applications, the standardization and commercialization of *software defined networking* (SDN) [2, 3], and the proposal of *network functions virtualization* (NFV) [4], where Cloud and SDN come from the datacom and both inspire the telecom and other third-party stakeholders, e.g. many *over-the-top* (OTT) service providers in 5GPPP, to propose new innovations such as NFV for core networks and software defined Cloud *radio access networks* (C-RAN) [5].

These innovations can meet differential QoS needs in three major aspects, such as *extreme mobile broadband* (eMBB) for *video content delivery* (CDN), *ultra-reliable low-latency communication* (uRLLC) for advanced connected vehicles [6], and *massive machine-type communication* (mMTC) for *internet of things* (IoT). However, these applications also impose technically challenging issues, such as flexible function split of base stations or small cells for C-RAN [7], software defined fronthaul and backhaul networks [8], customized multi-tenancy service/network slicing of core-network resources, and mobile edge deployment of time-critical cloud applications.

To address the aforementioned issues, NFV plays a pivotal role, where customized chaining of desired *virtual network functions* (VNFs), equivalently *service slicing* or *network slicing* [9, 10], can be demanded for differential use cases, such as Network Store [11]. Based on the levels of realization complications, the following use cases are expected in sequential phases: (1) vCPE for *virtual customer premise equipment*, remotely installable and maintainable by the central office to enable the simplicity of customization and management of user-demanded VNFs; (2) vEPC for *virtual evolved packet core*, to achieve multi-tenancy-based core network slicing and logically independent administration, allowing for value-added OTT VNFs; (3) vRAN for *virtual radio access network*, to realize a *virtual base-band unit* (vBBU) *pool* for C-RAN or FlexRAN [12] management.

This paper presents a study on Docker container [13, 14] based virtualization design of the vEPC, where the ultimate goal is to containerize all the component functions of the EPC [15] such that each component function becomes a

VNF and customized chaining of these VNFs plus extra value-added OTT VNFs can be formed. In other words, a customized chain of VNF containers can thus be viewed as a *network slice* or *service slice*. In this paper, a system design on how to *containerize* and thus *virtualize* a *light-weight EPC* (LW-vEPC) *slice* is presented, in order to be deployable at a mobile edge cloud, as aforementioned. In addition, the allocation methodology of multiple CPU cores for such a LW-EPC slice is being developed. In this paper, an algorithm called *specifically assigned cores* (SAC) is also proposed to achieve better utilization of CPU cores. Our preliminary results show that SAC outperforms the default scheme, namely *randomly assigned cores* (RAC).

The remainder of this paper is organized as follows. Section II details the system design of a vEPC slice, and its benefit to be deployed as a LW-vEPC slice near the mobile edge. Section III describes the proposed SAC algorithm and explains its design principles. Section IV demonstrates the function test of a vEPC slice consisting of two separate containers, and the performance evaluation of SAC, compared to RAC. Concluding remarks and outlook to future works are summarized in Section V.

II. Proposed System Design of vEPC and LW-vEPC

The realization of the whole system design is based on an open-source software package for 4G/5G called OpenAirInterface (OAI) [16]. One major contribution of this paper is to introduce the design and implementation details of component VNFs for vEPC using the Docker container technology, and discuss the benefit of a LW-vEPC slice at the mobile edge.

A. vEPC Slice Design with Two Containers

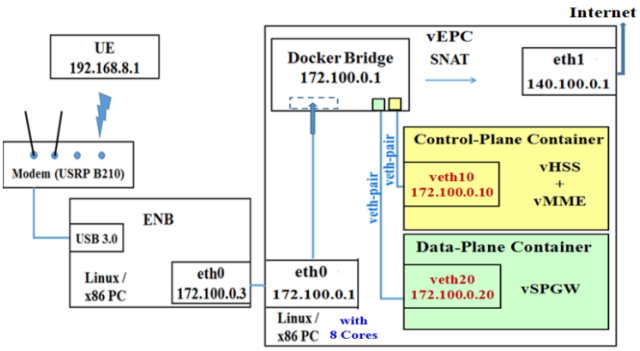

Fig. 1 System design of a vEPC with CP and DP containers

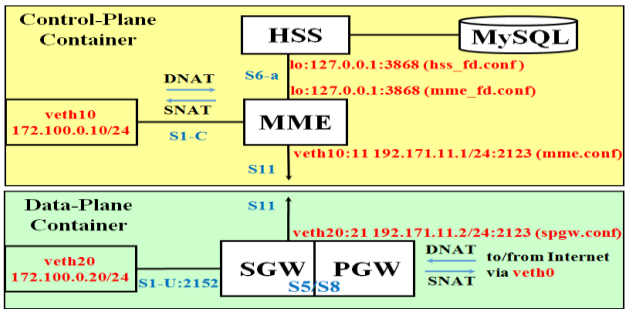

Fig. 2 Function blocks of CP and DP containers

The proposed system design of a vEPC slice with two Docker containers is detailed in Figs. 1 and 2. Fig. 1 gives the entire view of the system, where it is clearly seen that a vEPC slice, chained with a *control-plane* (CP) container and a *data-plane* (DP) container via a Docker Bridge using two *veth-pairs*-based virtual links, is created on top of an x86 PC of 8 CPU cores running with Linux, equipped with two *network interface cards* (NICs), where the first NIC (*eth0*) is connected with a small-cell base station (ENB) with a *software-defined-radio* (SDR) modem (USRP B210) for the radio access of user equipment (UE), and the second NIC (*eth1*) is connected to the Internet. Note that it is required to insert the first NIC (*eth0*) into the Docker Bridge for the bride to function correctly. In addition, an IP-masquerading mechanism via *source network address translation* (SNAT) is needed for access to the Internet.

Fig. 2 details the function blocks of both the proposed CP and DP containers. Based on the virtualization technology of Docker containers, the CP container *virtualizes* both the control subsystem functions: (1) *home subscriber server* (HSS) connected with a MySQL database for UEs' SIM account management and association authentication, and (2) *mobility management entity* (MME) for UEs' movement handover between ENBs. Note that, as of the writing of this paper, the handover function is not implemented in the OAI yet. By the same token, the DP container *virtualizes* the data-plane subsystems functions: (1) serving gateway (SGW) for establishing data tunneling between UE and EPC based on the GPRS tunneling protocol (GTP) for the sake of security and providing internal routing among ENBs, and external routing to the Internet via the S5/S8 interface; (2) *packet data network gateway* (PGW) for routing uplink and downlink traffic flows to/from the Internet and for providing dynamic IP addressing to UEs, and establishing the EPS bearer for QoS requirements. Note that the S5/S8 interface between SGW and PGW is not implemented for simplicity, and thus they can be denoted as SPGW throughput this paper.

Once virtualized, the CP container can perform as vHSS and vMME, and the DP container as vSPGW. However, the main challenging tasks for this study are two folded: internal linking within each container (denoted as *intra-container linking*) and external linking between the two containers (denoted as *inter-container linking*). For *intra-container linking*, the main design question is how to form multiple internal IP-domains based on a single virtual port equipped with each container. The answer is the *binding* mechanism provided by the Linux NIC interface, even if the interface is a virtual one. In other words, *binding* allows a single NIC interface to be associated with multiple IP addresses via a pre-definition in the default NIC configuration file under the /etc directory. For instance, the MME virtual port *veth10:11* is associated with *veth10* via *binding* such that they share the same entry/exit of the CP container. Similarly, the SGW virtual port *veth20:21* is associated with *veth20*. Note that the default network configuration files (*.conf in Fig. 2) should be appropriately modified for each individual component function according to the above new configurations. For *inter-container linking*, on the other hand, the main design questions are two folded: (1) how to resume the S11

interface between MME and SGW between the CP and DP containers, and (2) how to resume the external connections of both MME and the SPGW. For MME, the answer is that the CP container establishes both *source* and *destination* NAT (SNAT and DNAT) rules in the NAT table (i.e., the *-t nat* option when using the *iptables* toolset) for *outbound* and *inbound* flows, respectfully for *masquerading* of source IP-addresses and *port-redirection* (or *port-mapping*) for a specific internal application. For PGW, the answer is similar. In addition, SGW needs a specific *iptables* rule for the GTP tunnel traffic. The function test results are presented in Figs. 4 and 5.

B. LW-vEPC Slice Design and Fast Provisioning

The aforementioned design can further be developed as a LW-vEPC slice, in the sense that the DP container can be deployed near the mobile edge, or even co-located with the ENB or the vBBU pool of C-RAN. This is beneficial to those time-critical applications such as the communication of connected vehicles or the control of flying drones. With the help of a mobile edge cloud, bandwidth-consumptive CDN services can also be redirected to the edge cloud to save the *outbound* bandwidth and latency.

In the case of no handover requirement, the design of the DP container with vSPGW should be able to function and survive. However, if the handover between ENBs is required, the DP container should be redesigned to contain vMME. However, this is currently out of the capability of OAI because the OAI EPC software cannot handle handover yet.

In terms of VNFs' provisioning, the Container-type virtualization is famous with its powerful *imaging* ability in the sense that once an image is provisioned for the desired container functions, the image can fan out many containers of the same functions very fast, which makes the provisioning time much shorter than the Hypervisor-type virtualization, namely the level of *seconds* to *minutes* versus the level of *hours* to *days*. The fundamental difference between the two virtualization technologies lies in that the Hypervisor-type is based on *guest-OS confinement*, while the Container-type on *process confinement*. Thus, the latter consumes much lighter-weight resources than the former, and is more agile to differential customization and more adaptive to traffic load variations. Thus, these justify the adoption of the Docker container for provisioning flexible and real-time deployment of VNFs.

III. PROPOSED ALGORITHM

Efficient allocation and orchestration methods of system resources among the vEPC slices or LW-EPC slices are under development. As aforementioned, this paper proposes a *simple* but *powerful* algorithm, called *specifically assigned cores* (SAC), in better saving the resource and boosting higher performance of CPU cores than the default RAC algorithm. Such an issue is important since the multi-core technology is the main stream of modern CPUs. Furthermore, saving the use of CPU cores is also energy-green, in particular when running many VNFs or Containers in a computing farm such as a cloud-based data center. The SAC algorithm adopts two design principles as listed below.

- Principle 1 (Bottleneck Identification)
  It is obviously important to identify the bottleneck of the system performance between the DP and CP containers such that the dominant one deserves more CPU resources. Generally speaking, the DP traffic dominates the CP signals in terms of bandwidth demands, thus it seems to be justified to assign more CPU resources to the DP containers. This is quite common to most of the 5G use cases. However, in the case of mMTC, where massive machine-type communications may involve huge amount of control signals during their connection associations with the EPC, and thus the CP signals may well dominate over the DP traffic. In such a case, the CP container deserves more CPU resources instead.
- Principle 2 (CPU Core Pinning)
  When conducting a resource allocation of CPU cores, the allocation unit is important, which could be *integral* or *fractional*. The *integral* allocation unit is obviously more convenient and simpler to a computing farm with a lot of CPU cores, and not so friendly to a physical machine with only a limited number of CPU cores such as a generic PC. Although the *fractional* allocation unit is obviously more flexible and friendly the use of CPU cores, it requires the support of the adopted virtualization technology. To the best of the authors' knowledge, the Container type such as Docker or LXC can support both, but the Hypervisor type cannot support the *fractional* one yet. Again, this justifies the adoption of the Docker container for this study. Various allocation methods in combining and leveraging both are underway in our lab. However, for simplicity, this paper only focuses on the *integral* one to demonstrate the power of the proposed SAC algorithm, which emphasizes on *avoiding unnecessary context-switching overheads* among the allocated CPU cores to a vEPC which can be expected to be worse as the traffic load level increases.

As a simple illustration, according to Principle 1, the proposed SAC algorithm was firstly exercised for the aforementioned LW-vEPC slice (equivalently, the DP container with vSPGW only) considering this use case where the DP traffic dominates the CP signals is of the most common and highest interest, on a generic x86 PC with 8 CPU cores, equipped two physical Gigabit Ethernet (GbE) NICs and a PCI-E mother board. As aforementioned, only the *integral allocation unit* was considered to simplify the discussions. Given the simplest allocation scenario where a single CPU core is demanded for a LW-vEPC slice, formed by the DP container with only vSPGW, the proposed SAC algorithm *specifically assigns* one CPU core to avoid *unnecessary context-switching overheads*, according to Principle 2, with a special trick on specifying the largest CPU core ID to avoid any possible random selection from the smallest ID. Of course, given the fact that the default RAC scheme with a periodic load-balancing capability from the Linux kernel for *symmetric multiprocessing system* (SMP), performed every 200 ms, this consideration might become trivial. However, a shorter term than this period with traffic bursts might generate some benefit due to this trick with the SAC algorithm.

IV. Function and Performance Tests

This section gives the function test of a vEPC slice of two containers, namely the CP and DP containers, to illustrate the successful verification of the system design and the linking capability of *intra-container* and *inter-container* communications, as depicted in Figs. 1 and 2 and detailed in Section II. In addition, some preliminary results for performance comparison between the proposed SAC algorithm and the default RAC algorithm is given for demonstrating the powerful design of SAC.

A. Function Test

Fig. 3 presents a joint system function test of a vEPC slice of two containers (i.e., the CP and DP containers) with the ENB, allowing for the UE (cell-phone) connection association to receive a downlink video streaming from the Internet successfully. As previously explained in Fig. 1, the x86 PC on the left, running with a low-latency Linux kernel (v3.19), hosts the OAI ENB software and performs as a ENB, together with the SDR modem (USRP B210); the x86 PC on the right displays the operations of vHSS, vMME and vSPGW in a top-down fashion, where vHSS and vMME is formed by the CP container and vSPGW by the DP container.

This above success also verifies the answers to the main design questions raised in Section II, for both the *intra-container* and *inter-container* communications. Fig. 4 lists all the Kernel-based and Docker-based NAT chains, set with SNAT and DNAT rules, within the CP container. Note that the DOCKER_OUTPUT chain, defined on top of the Kernel OUTPUT chain, is in charge of DNAT for both TCP and UDP *inbound* flows to find the CP container itself (127.0.0.11) when arriving at its only entry (*veth10*, recalled from Fig. 2) using a randomly pre-set internal port-redirection for TCP (34245) and UDP (37114). On the other hand, the DOCKER_POSTROUTING chain, defined on top of the Kernel POSTROUTING chain, is responsible for SNAT for both TCP and UDP *outbound* flows to *masquerade* their source IP address and port number. Note that the masqueraded port number 53 may confuse the outside world with that these outbound flows were from a DNS server, but actually they are not. Fig. 5 also lists all the Kernel-based and Docker-based NAT chains, set with SNAT and DNAT rules, within the DP container, where both the DNAT and SNAT behaviors work similarly, respectively at the DOCKER_OUTPUT and DOCKER_POSTROUTING chains. A major difference is that the UE's uplink data flow (i.e., non-control flow from 192.188.0.0, identified by non-SCTP or !sctp) should also be specifically masqueraded with the IP address 172.100.0.20 of the egress port of vPGW (*veth20*, recalled again from Fig. 2), and eventually masqueraded again at the egress port (*eth1*) of the physical machine with 140.100.0.1 for access to the Internet.

B. Performance Evaluation and Analysis

Good understanding of the CPU response to the external traffic load can be helpful to the design of CPU resource allocation. Particularly, for a multi-core CPU, such understanding is complicated and thus challenging. For this study, two types of test results are presented. Type-1 test is

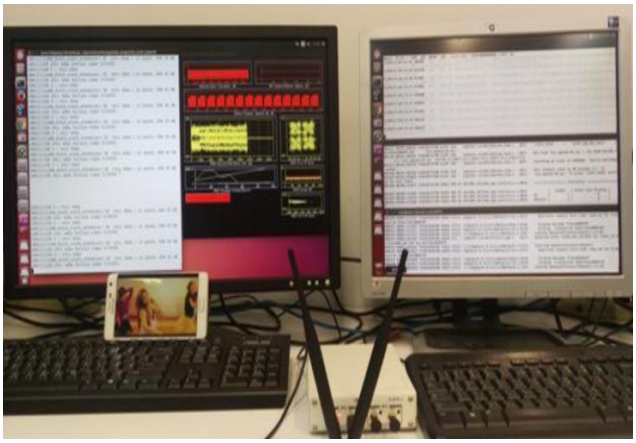

Fig. 3 Joint system function test of a vEPC slice of two containers (the CP and DP containers) with the UE/ ENB

```
root@hss-mme:/# iptables -L -t nat

Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target                  prot opt source     destination
DOCKER_OUTPUT           all  --  anywhere   127.0.0.11

Chain POSTROUTING (policy ACCEPT)
target                  prot opt source     destination
DOCKER_POSTROUTING      all  --  anywhere   127.0.0.11

Chain DOCKER_OUTPUT (1 references)
target     prot opt source               destination
DNAT       tcp  --  anywhere             127.0.0.11          tcp  dpt:domain to:127.0.0.11:34245
DNAT       udp  --  anywhere             127.0.0.11          udp dpt:domain to:127.0.0.11:37114

Chain DOCKER_POSTROUTING (1 references)
target     prot opt source               destination
SNAT       tcp  --  127.0.0.11           anywhere            tcp  spt:34245 to::53
SNAT       udp  --  127.0.0.11           anywhere            udp spt:37114 to::53
```

Fig. 4 SNAT/DNAT iptables rules (CP container)

```
root@spgw:/# iptables -L -t nat

Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target                  prot opt source     destination
DOCKER_OUTPUT           all  --  anywhere   127.0.0.11

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
SNAT       !sctp --  192.188.0.0/24      anywhere            to:172.100.0.20

Chain DOCKER_OUTPUT (1 references)
target     prot opt source               destination
DNAT       tcp  --  anywhere             127.0.0.11          tcp  dpt:domain to:127.0.0.11:42069
DNAT       udp  --  anywhere             127.0.0.11          udp dpt:domain to:127.0.0.11:59329

Chain DOCKER_POSTROUTING (0 references)
target     prot opt source               destination
SNAT       tcp  --  127.0.0.11           anywhere            tcp  spt:42069 to::53
SNAT       udp  --  127.0.0.11           anywhere            udp spt:59329 to::53
```

Fig. 5 SNAT/GNAT/GTP iptables rules (DP container)

for studying the CPU response behavior *with full cores* to the external traffic load on a *physical machine* (PM), as illustrated in Fig. 6, where a generic x86 PC with 8 cores (actually *hyper-threaded* from 4 physical cores) was adopted. On the other hand, Type-2 test is for studying the CPU response *with allocated cores* to the incoming traffic load to a container, as illustrated in Fig. 7, where the DP container demands a single core, allocated by RAC and SAC.

The major messages delivered by Fig. 6 are listed below:

- *Idle* response phase (traffic < 20 Gbps)
  During this phase (CPU is idle), the average CPU load over 8 cores is still *light* enough, i.e., < 20%. Equivalently, 20% (average) can be scaled to 160% (total) if the total CPU limit 800% is considered. Note that the response is somewhat non-linear. Also note that the *packet loss rate* (PLR) is either zero or invisible.
- *Linear* response phase (traffic ∈ [20, 80] Gbps)
  During this phase (CPU can work normally), the average CPU load over 8 cores linearly increases with the UDP test flow rate. The *linear* phase allows for a predictable response when the container-based VNFs demand some resources of the CPU cores from the PM host. Note that the PLR is still either zero or small.
- *Saturated* response phase (traffic > 80 Gbps)
  During this phase (CPU is over busy), the CPU load shows descending or saturating slopes as the traffic rate increases. This suggests that the PM host cannot longer provide the desired core performance, and more demands from the container-based VNFs should not be admitted any longer or should be redirected to other worker PMs if they are existent and can permit the demands. This is also strongly suggested by the fact that the PLR starts to shoot up and become worse with more traffic.

One the other hand, the major messages delivered by Fig. 7 are also listed below:

- *Much shorter linear* response phase (traffic < 20 Gbps)
  The linear response phase is much shorter than that of Fig. 6. This is highly expected since only a single core is allocated to the DP container, either via RAC or SAC. Since the linear phase of the CPU load is short enough, the PLR becomes a much better metric to the upper end of this phase, say 20 Gbps in light of the shooting up of the PLR here. Obviously, the *linear* phase is also expected to be longer if more CPU cores can be allocated.
- *Very early saturated* response phase (traffic > 20 Gbps)
  The saturated response phase, *early* and *long*, has a different operational meaning than the one in Fig. 6, and can thus provide a good competition playground for performance comparison of various allocation strategies. In other words, the demanded container should try to *make the best use of the assigned core* even if the assigned core starts to saturate so early. Depending on the level of QoS requirements, say PLR, one can still make some efforts in finding a better allocation algorithm leading to a lower CPU load response and less packet loss as well. It is clearly seen that the proposed SAC algorithm outperforms the default RAC algorithm, in terms of both *lower CPU load response* and *less packet loss*. In the case of PLR, such superiority amplifies with the traffic load, and this shows the true power of SAC, despite that its superiority in saving the CPU load becomes slightly less visible when the traffic > 40 Gbps, which suggests that even the SAC algorithm is getting too much CPU load and it is time to demand more CPU resources. Also note that, beyond 40 Gbps, the CPU load of RAC is more than 100%, which is possible because the demand from RAC is switched among different cores and these overheads can add up together to pass over the single-core limit 100%, averagely speaking.

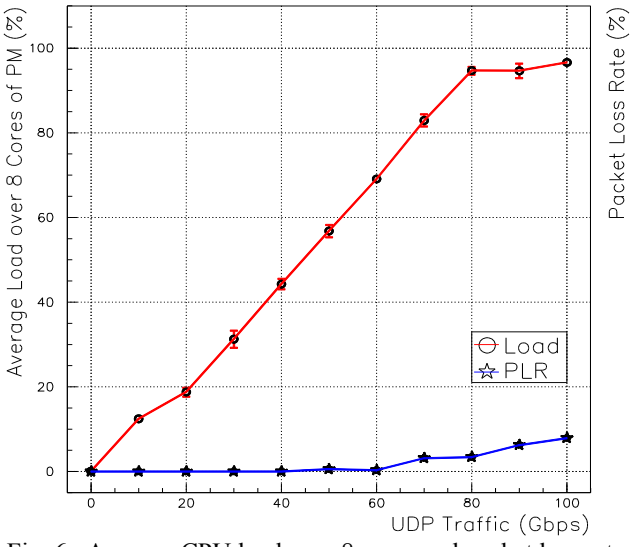

Fig. 6 Average CPU load over 8 cores and packet loss rate (PLR) versus the UDP test flow rate on the PM host

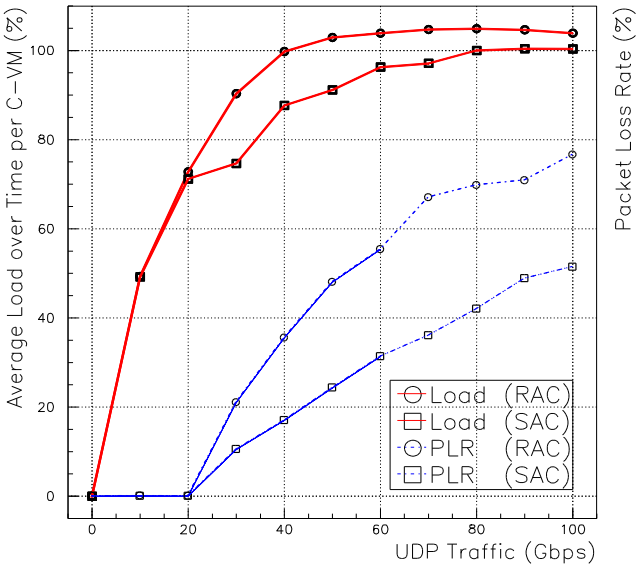

Fig. 7 Average CPU load over time and packet loss rate (PLR) versus the UDP test flow rate to the DP container (single-core allocated by the RAC and SAC algorithms)

V. Conclusion and Outlook

In this study, a system design of a vEPC slice with two Docker containers, namely the CP container with vHSS and vMME and the DP container with vSPGW, has been proposed and functionally verified. The demonstration results show a successful joint test of this vEPC slice with an ENB, which enables the UE to successfully receive a video streaming from the Internet. This success also verifies our design efforts in dealing with both *intra-container* and *inter-container* communication designs via the Docker bridge and in-container NAT rule setting (SNAT/DNAT) and other network configurations. In addition, a *powerful but simple* CPU resource allocation algorithm called *specifically assigned cores* (SAC) has also been proposed. Our preliminary results show that the understanding of a physical machine's response to external traffic level can provide a predictable linear design, serving a good design guideline for developing efficient resource allocation strategies for CPU resources. In addition, it has also been shown that the proposed SAC algorithm outperforms the default RAC algorithm in terms of *lower CPU load response* and *less packet loss*, and the performance gap of SAC over RAC amplifies with the traffic level. This is helpful in suggesting some further design guidelines on how to make the best use of the allocated CPU cores, and on when would be the appropriate time to demand more resources.

For future works, individual containerization and thus virtualization for each subsystem function of the EPC would also be interesting. For instance, containerizing each VNF (i.e., placing vSPGW, vMME and vHSS in three different containers) can increase the flexibility for customized chaining or slicing of VNFs, but would also increase the linking complexity. Thus, different linking capabilities of the Docker technology will be studied. In addition, an improved version of CPU resources allocation is also interesting, where the allocation unit cannot only be *integral*, but also *fractional*, or even *both*. To generate SAC-like improvement works, more extensive studies on the traffic dynamics interaction between the physical machine and the VNF containers it provides is also needed.

The ultimate goal of 5G core network virtualization [17] is moving toward a direction called *microservice*, which will further divide the core network functions into more specific micro-functions. In that scenario, how to virtualize and link these micro VNFs would be very challenging because the complexity of linking becomes so high, and the point-to-point linking will be too complicated to implement, and might also generate intolerable latency. In the case, a simple message bus such as *protobuf* can be expected to reduce the design complexity of *microservice*-based VNFs. Our future research will also move toward this direction.

Acknowledgment

This study was supported by the Taiwan Ministry of Science and Technology under research grant 107-2221-E-155-013.

References

[1] *View on 5G Architecture*, 5GPPP White Paper v2.0, Dec. 2017.
[2] D. Kreutz, P. E. Verissimo, and S. Azodolmolky, "Software-defined networking: a comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.
[3] Y. Li and M. Chen, "Software-defined network function virtualization: a survey," *IEEE Access*, vol. 3, no. 1, pp. 2542–2553, 2015.
[4] *Network Functions Virtualization* (NFV)*; Ecosystem; Report on SDN Usage in NFV Architectural Framework*, ETSI NFV-EVE White Paper 005, Dec.2015.
[5] N. Nikaein et al., "Towards a cloud-native radio access network," in *Advances in Mobile Cloud Computing and Big Data in the 5G Era*, vol. 22. 3, C. X. Mavromoustakis et al., Ed. Cham: Springer, 2017, pp. 171–202.
[6] *A study on 5G V2X Deployment*, 5GPPP White Paper v1.0, Feb. 2018.
[7] Commscope. What are C-RAN small cells [Online]. Available: http://www.commscope.com/Solutions/what-are-c-ran-small-cells
[8] Heli Zhang, Jun Guo, Lichao Yang, et al., "Computation Offloading Considering Fronthaul and Backhaul in Small-Cell Networks Integrated with MEC," *IEEE Conf. on Computer Communications Workshops*, Atlanta, GA, USA, May 2017, pp. 115–120.
[9] *Description of Network Slicing Concept*, NGMN White Paper, Jan. 2016.
[10] X. Foukas, G. Patounas, A. Elmokashfi and M. K. Marina, "Network slicing in 5G," *IEEE Comm. Magazine*, vol. 55, no. 5, pp. 94–100, May 2017.
[11] N. Nikaein et al., "Network store: exploring slicing in future 5G networks," *Proc. 10th Int. Workshop on Mobility in the Evolving Internet Architecture (MobiArch '15)*, Paris, France, Sep. 2015, pp. 8–13.
[12] X. Foukas, N. Nikaein, M. M. Kassem et al., "FlexRAN: a flexible and programmable platform for software-defined radio access networks," *Proc. 12th Int. Conf. on Emerging Networking Experiments and Technologies* (CoNEXT '16), Irvine, CA, USA, Dec. 2016, pp. 427–441.
[13] D. Bernstein, "Containers and cloud: from LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.
[14] I. Miell and A. H. Sayers, *Docker in Practice*. New York: Manning, 2016.
[15] *The journey to packet core virtualization,* Alcatel-Lucent White paper, 2014.
[16] N. Nikaein, M. K. Marina, S. Manickam et al., "OpenAirInterface: A flexible platform for 5G research, *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 5, pp. 33-38, 2014.
[17] *System Architecture for the 5G system; Stage 2*, 3GPP TS 23.501 v15.00, Dec. 2017.