

Enhancing CNF performance for 5G core network using SR-IOV in Kubernetes

Dinh Tam Nguyen* **, Ngoc Lam Dao*, Van Thuyet Tran*, Khac Thuan Lang*,
Thanh Tu Pham*, Phi Hung Nguyen*, Cong Dan Pham*,
Tuan Anh Pham*, Duc Hai Nguyen*, Huu Thanh Nguyen**

*OCS Research Center, Viettel High Technology, Viettel Group, Hanoi, Vietnam

**Hanoi University of Science and Technology, Hanoi, Vietnam

(tamnd17, lamdn10, thuyettv1, thuanlk)@viettel.com.vn

(tupt18, hungnp22, danpc)@viettel.com.vn

(anhpt6, haind13)@viettel.com.vn, thanhnh@hust.edu.vn

Abstract—Cloud-native containerization is replacing Virtual Machine (VM) as the preferred method of application deployment. However, the network requirements involved, including high performance and low latency, prove to be a barrier to the deployment of fifth-generation (5G) telecommunication applications. Cloud-native Function (CNF) enables network functionalities to run in containers instead of on traditional baremetal servers. The success of integrating Single Root Input/Output Virtualization (SR-IOV) into Network Function Virtualization (NFV) on VMs has motivated our research into using SR-IOV in CNF, in order to address the strict network demands. When applied to the Charging Function (CHF) in the Cloud-native 5G architecture, our solution resulted in a 30% increase in network throughput, compared to the Kubernetes deployment using Calico. Therefore, SR-IOV and CPU-Pinning have shown potential in optimizing the performance of Cloud-native container-based applications.

Keywords—CNF, SR-IOV, Performance, 5G, Cloud-native

I. INTRODUCTION

Kubernetes is a container-based deployment platform with many benefits such as elasticity, durability, ease of management, and cost-effectiveness. A survey by the Cloud Native Computing Foundation (CNCF) [1] [2] showed that the respondents using Kubernetes in production increased from 58% in 2018 to 83% in 2020, implying container-based deployment has become the trend and surpassed VM as well as bare-metal deployments. On the other hand, the 1:n pod mapping architecture employed in Kubernetes's service discovery exacerbates the delays caused by container processing, as well as the overhead of sharing resources (CPU, I/O, Networking, etc.) between containers [3] [4]. Both the delays and the computing overhead may compromise the ultra-low latency in 5G.

Network Function Virtualization (NFV) with Single Root Input/Output Virtualization (SR-IOV) has been documented to provide better network performance on VMs [5]. Thanks to SR-IOV, multiple operating systems can share a physical interface via Virtual Functions (VFs) on the same Network Interface Card (NIC) [6], which maximizes utilization of resources such as I/O devices [7].

5G is set to cater both to a myriad of new services (such as AR/VR, Mobile Edge Computing) and to traditional services (such as Voice, Data, and SMS). An inefficient charging system would be the bane of a network meant for an ever-escalating number of transactions. Currently, we are using a real-time cost calculator called Viettel Online Charging System, or vOCS for short, to track telecommunication service usage. The current version, vOCS 3.0, is running on bare-metal servers without enhanced network solutions such as Calico and SR-IOV. The performance required of 5G has rendered our current bare-metal deployment glaringly unfit while making cloud migration a more sensible option for the next version, vOCS 4.0 for 5G. To leverage the cloud architecture, the 5G core relies on modularized and containerized cloud-native network functions, which enable highly flexible scaling and function lifecycle management [8]. Currently, 5G Core products offered by notable vendors such as Ericsson [9], Samsung [10] and Huawei [11] are built on cloud-native, microservice-based technology. Yet, the available literature hardly explores how different cloud native architectures affect the performance of 5G Core.

In the realm of virtualization, performance of different network models has attracted considerable interest [12] [13] [14] [15]. Some researches have investigated this topic specifically in the context of Kubernetes deployment. For instance, Kumar and Trivedi [16] noted that Calico maintained adequate bandwidth for transmitting packets with low resource utilization. Therefore, we have decided to test Calico against SR-IOV, another Container Network Interface (CNI) plugin reported to have cut down virtualization-induced bandwidth loss by 2% and latency by 10% in VMs [17]. Expanding on results regarding SR-IOV, Jie Zhang [18] compared bandwidth utilization and latency experienced by VMs with SR-IOV, VMs with PCI passthrough, and containers with PCI passthrough. Nevertheless, the article's comparisons left out containers with SR-IOV and the authors did not comment on resource consumption. Our paper serves to address this lack, as well as the paucity of published works that detail different

cloud-native deployments' impact on 5G Core's performance.

Aiming to emulate the success seen in VMs using NFV with SR-IOV, we implement SR-IOV in cloud-native network-intensive 5G applications. Our test was undertaken within the Charging Function (CHF), a module in what is to be vOCS 4.0 for 5G. CHF is an apt candidate for the purpose of our research because it requires the lowest latency for a smooth-running vOCS. Our solution using SR-IOV and CPU-Pinning has boosted the network throughput by 30% in comparison with the Calico-enabled Kubernetes deployment.

The rest of the paper is as follows. Section II details the three network models for CHF that we have evaluated: the bare-metal model, the Cloud-native model using Calico, and the Cloud-native model using SR-IOV. Section III delves into the optimization technologies and techniques involved, such as the microservice chassis library, CPU-pinning, before laying out the architecture of the CHF. Section IV delineates our testing methodology as well as our evaluation criteria. Finally, section V offers some insight into the results and then ponders directions for future research.

II. ARCHITECTURE

A. Bare Metal Model

In this network model, all the components of vOCS run on the same operating system, sharing physical resources like CPU, RAM, and Network. This model does not suffer from the extra overhead caused by virtualization; however, heavy workload and frozen applications can detrimentally impact the system as a whole, leading to degradation in quality of service.

B. Cloud-native Calico Network Model

Kubernetes offers the option to integrate different network solutions through CNI plugins. A CNI plugin inserts a network interface into the container's network namespace and makes necessary changes to the network configuration on the host. The plugin also assigns an IP address to the interface and sets up routes that are consistent with the IP Address Management section by invoking the appropriate IP Address Management (IPAM) plugin [19].

Calico, a common CNI, not only creates overlay networks and establishes connections between pods across the nodes, but also manages network security and administration. Figure 1 visualizes the Cloud-native Model using Calico. The Calico node agent consists of two main components: Felix and BIRD. Felix, running on machines that host endpoints, ensures that only valid traffic crosses endpoints by programming routes and iptable on the host. BIRD is the BGP (Border Gateway Protocol) daemon that distributes routing information to other nodes. Calico supports three routing modes: Native, IP-in-IP, and VXLAN. The network model using Calico applies the IP-in-IP encapsulation type only to traffic crossing a subnet boundary. The rationale behind using Calico in our testbed was that it offers a satisfactory bandwidth while not prone to resource exhaustion [16].

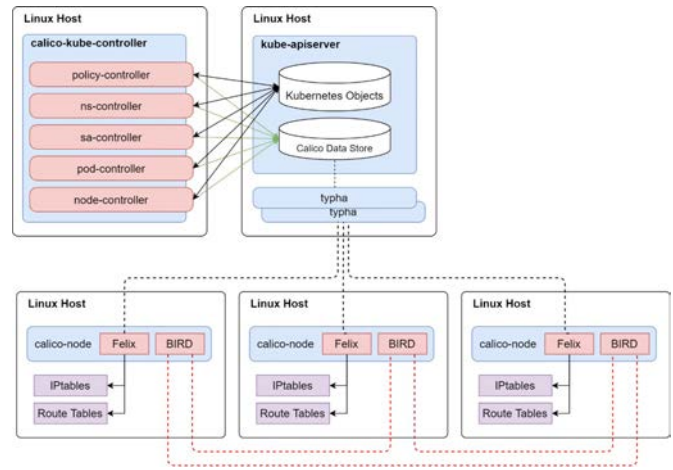


Figure 1. Calico architecture

C. Cloud-native SR-IOV Network Model

Similar to Virtual Machine [20], containers on each worker node are sharing a virtualized single port of the NIC via software emulation. Figure 2b shows the container-based network model, with the container's network port connecting to NIC via Container Network Interface and Container Network Plugin. The multiple layers of emulation predictably worsen container overhead [21]. Furthermore, surging network traffic is a burden to the CPU, taking up computing resources meant for other applications.

SR-IOV enabled physical network functions to be shared without software emulation. It provides two function types: The Physical Function and Virtual Function. The Physical Function, a PCIe function of a SR-IOV compatible NIC, configures and manages the SR-IOV functionalities, such as enabling virtualization and exposing Virtual Functions. Directly accessible to the operating system, each Virtual Function represents a virtualized instance of the NIC and has its own PCI configuration space [20] [22]. Virtual Functions share physical resources on the NIC with each other and with the Physical Function. Thanks to SR-IOV, the container can directly send and receive packages to/from Virtual Functions in the user space instead of the kernel space. Taking CNIs out of the equation also removes the associated computing cost and latency.

Figure 3c illustrates SR-IOV being applied to Kubernetes containers, which skipped the worker node's kernel space and directly connected to VFs via a physical NIC. Calico CNI was used for management network in this case. Obviating CNIs saved computing resources because the network traffic was processed in the NIC and because the simulation of software package processing did not incur extra computational cost. Furthermore, latency was also mitigated because the network traffic had to deal with fewer processing layers.

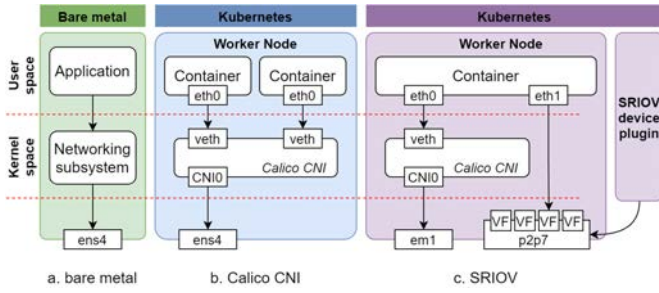


Figure 2. Network models for vOCS

III. IMPLEMENTATION

A. Microservice chassis

Kubernetes implements the decoupling architecture in which services are separated from each other. The connections between them are established via service discovery [23] as well as internal Domain Name System (DNS). These network interactions contribute to longer request and response times as well as higher latency overall. A Service of *LoadBalancer* type is attached to an Ingress Controller, a set of Ingress that defines where to forward traffic. Each traffic to a service container that passes through the Ingress controller aggravates the container's network connectivity overhead.

The *microservice chassis* library alleviates this issue as it lets applications discover the addresses of service containers and connect to them directly. The chassis accommodates scaling by updating the set of container addresses. As a result, network latency should be adequately lessened for telecommunication applications.

B. CPU-Pinning

CPU scheduling algorithms in operating systems elect a process to use the CPU at a given time while the other processes are put on hold. The wait time can accumulate into discernable latency when the operating system has to handle many processes, including application processes or containers. In the case of a heavy workload, the operating system's native CPU scheduling may jeopardize the performance of 5G Core, whose processes may have their resources temporarily taken away.

To fix this problem, CPU-pinning can be used to manually assign a fixed set of CPU cores to a process or a container. The process or container then owns a fixed number of vCPU in the course of its whole lifecycle and no longer has to wait for others, which is conducive to enhanced performance.

C. Charging Function (CHF)

The 5G network should be able to support a large number of devices, various types of data, and widely different bandwidth requirements. Accordingly, the traffic simulator in our test has to simultaneously generate multiple services of various types, so that the test scenario is as close to the actual production setting as possible.

A crucial module of 5G core, the cloud-native CHF enforces precise 5G network control via complex policies based on each service's attributes and its subscribers. Our test focused on the performance of the CHF.

The CHF has three layers:

- Connectivity: stateless sets of components that interface with external entities via an API gateway.
- Business logic: a stateless set of application layers that include business logic, policy engine and other necessary services.
- Data management: the data access layer handling multiple backend storages (such as Database).

The deployment architecture for CHF in vOCS 4.0 is shown in Figure 3:

- Connectivity layer:
 - Load balancer: an external-facing physical load balancer that receives traffic from other 5G services.
 - CGW: a gateway that receives incoming traffic via load balancer then converts it to a usable format.
- Business logic:
 - OCP: a component that handles calculation tasks.
- Data management:
 - ABM: a proxy that processes queries into the database.
 - Database: two Aerospike active-active database clusters, each containing information of 10 million subscribers.

IV. SIMULATION AND TEST

A. Methodology

We implemented the CHF with four different network models:

- The bare metal model, CPU-pinning not applied
- The bare metal model, CPU-pinning applied (the evaluation baseline)
- The Cloud-native model using Calico, CPU-pinning applied
- The Cloud-native model using SR-IOV, CPU-pinning applied

All implementations of the CHF followed the same deployment architecture, as shown in Figure 3

The test client generated the charging traffic to the CHF network charging gateway (CGW) pods. The traffic included both balance inquiries and a mixture of *initiate*, *update* and *terminate* charging operations across data, voice as well as messaging services.

TABLE 1. TRAFFIC PROFILE

Traffic type	Description
Voice	Initiate, Update and Terminate operations (180 seconds duration, 120 seconds between operations)
Data	Initiate, Update and Terminate operations (45 minutes duration, 120 seconds between operations)
SMS	Terminate operation

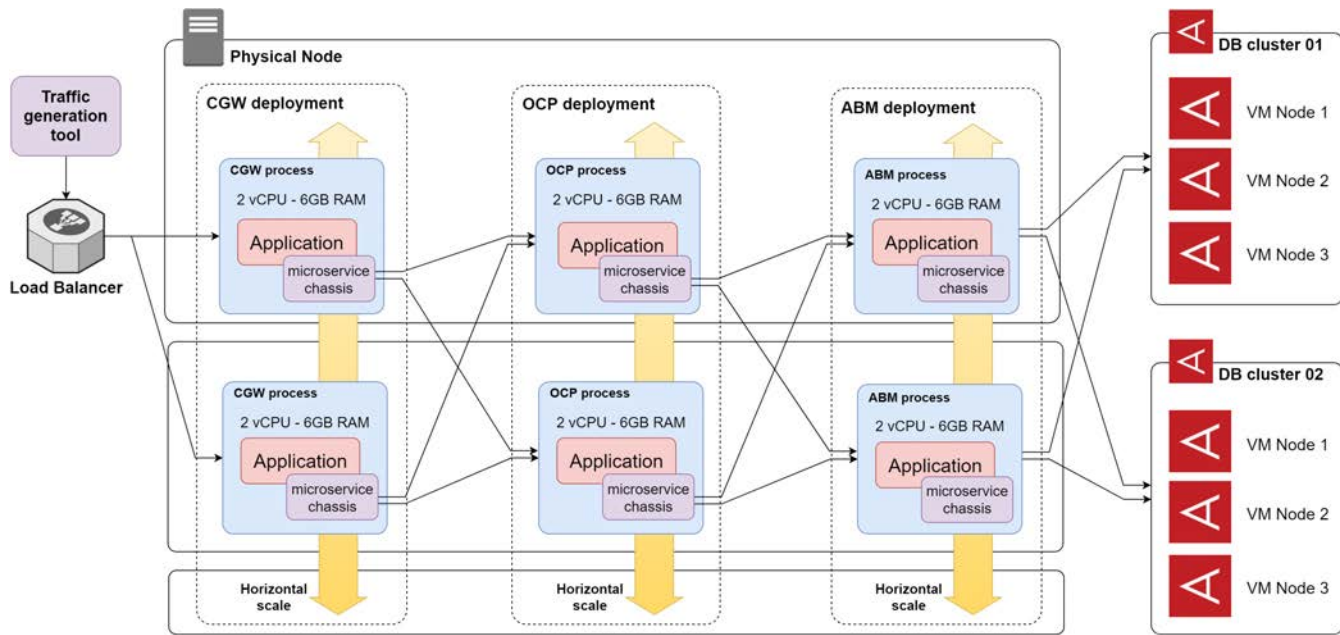


Figure 3. CHF architecture

Containers were run on three physical nodes with Intel 8260, which have ninety-six 2.4GHz threads and 256GB RAM. Each pod used two vCPU and 6GB RAM. Horizontal Pod Autoscaler was used during the entire test. The Kubernetes version was 1.18, and the Aerospike version was 4.13 in all deployments. Twenty million simulated accounts were provisioned over six Aerospike nodes.

Three metrics were considered: throughput (in TPS - transaction per seconds), latency (in milliseconds) and CPU utilization (in %).

B. Test results

Resource usage and system performance among different network models are listed in Table 2.

TABLE 2. PERFORMANCE RESULTS

	Bare metal model (free CPU)	Bare metal model (applied CPU pinning techniques)	Cloud-native Calico Network Model	Cloud-native SR-IOV Network Model
CPU Consume	40%	46%	39%	45%
TPS	50000	67000	54000	70000
Latency	2.93 ms	2.9 ms	2.94 ms	2.87 ms

The result in Table 2 shows that CPU-pinning compounded the CPU workload. In both the bare-metal model and the Cloud-native model using SR-IOV, CPU utilization reached 46% and 45% respectively. Meanwhile, the two remaining models were slightly more resource-conscious, approaching approximately 40% in CPU utilization.

The result also confirms the synergy between CPU-pinning and SR-IOV. In fact, the Cloud-native model using SR-IOV and CPU-pinning achieved 70,000 TPS, 1.5 times the TPS obtained by the bare-metal model without CPU-pinning (only 50,000 TPS). Therefore, this network model may help vOCS

serve significantly more customers. Interestingly, the latency did not go up but rather dropped to 2.87 ms. That said, the differences in latency among the four models were negligible.

In conclusion, the model using SR-IOV and CPU-pinning outperformed the bare-metal model with CPU-pinning by 30% in terms of network throughput and package processing. On the other hand, the Cloud-native model using Calico performed worse than the bare-metal model with CPU-pinning, while the bare-metal model without CPU-pinning performed the worst. The subpar performance of the model using Calico can be explained by the need for more hardware resources and time to encapsulate packets and route them to network cards. Meanwhile, SR-IOV helped containers directly connect to NIC; packets therefore bypassed the kernel stack in order to deliver high performance.

V. CONCLUSION

The optimistic results informed our decision to employ SR-IOV and CPU-pinning in the production deployment of vOCS 4.0. Besides SR-IOV, DPDK is gaining traction as a means to improve computing efficiency and packet throughput [24]. DPDK consists of libraries to accelerate packet processing by offloading TCP packet processing from the operating system's kernel to processes running in the user space. This capability has prompted us to tinker with DPDK, beside SR-IOV, in the pursuit of a better vOCS. We will also look into the optimal disk I/O for Kubernetes with high IOPS applications.

ACKNOWLEDGMENT

We want to express our gratitude to Viettel High Technology Industry Group and Hanoi University of Science and Technology for providing us with computing equipment and research resources.

505

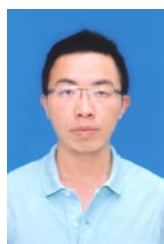
- ICACT2022 February 13 ~ 16, 2022



Phi Hung Nguyen (born 1996, in Hanoi, Vietnam) received his B.A. degree in computer science in 2018 from Franklin & Marshall College, Lancaster, Pennsylvania, USA. Since 2019, he has been working as a data analyst at Viettel High Technology Industries Corporation (VHT) of Viettel Group. His research interests revolve around distributed machine learning in telecommunications.



Cong Dan Pham was born in 1985 in Vietnam. He received his PhD. Degree in Probability and Statistics from Aix-Marseille University, France in Jun 2014. He has academic experience in universities like Aix-Marseille University, Hanoi University of Education, and industry experience companies like Viettel Group. He has published various journal and conference papers in IEEE and ScienceDirect. His research interests include probability and statistic, data science and machine learning.



Tuan Anh Pham was born 1985 in Vietnam. He received his B.Sc. in Computer Science Engineering from Hanoi University of Science and Technology, Vietnam in 2008. He is currently deputy director of the Research Center for OCS (Online Charging Platform) of Viettel High-Tech Industry Corporation (VHT). He has 13 years of experience in research and development of telecommunications core systems. His research focuses on developing technology platforms for telecommunications core networks such as 5G, Cloud native.



Duc Hai Nguyen (born 1986, in Vietnam) received his B.Sc. in Mathematics and Informatics Engineering from Hanoi University of Science and Technology, Vietnam in 2009. He is currently director of the Research Center for OCS (Online Charging Platform) of Viettel High-Tech Industry Corporation (VHT), Viettel Group. He has many years of experience in research and development of 5G core systems such as online charging system. His research interests include telecommunications core networks, Cloud Computing.



Huu Thanh Nguyen (born 1971, in Vietnam) received B.S and M.Sc in Electrical Engineering from Hanoi University of Science and Technology, Vietnam in 1993 and 1995, respectively. In 2002, he received his PhD with summa cum laude in Computer Science from the University of Federal Armed Forces Munich (Germany). From 2002 to 2004 he has been with the Fraunhofer Institute for Open Communication Systems (FOKUS) in Berlin, Germany. From 2004 until now Nguyen Huu Thanh has been working as associate professor in the

School of Electronics and Telecommunications, Hanoi University of Science and Technology, Vietnam. His research interests include radio resource management in next generation wireless networks, QoS/QoE, network security, the Future Internet, energy-efficient networking and software-defined networking. He is the leader and member of several national and international research projects on network security and the Future Internet. He is the former Chair of IEEE Vietnam Section (2013 - 2018), Vice President of the Radio Electronics Association of Vietnam (REV).