2015 IEEE 7th International Conference on Cloud Computing Technology and Science

Minimizing Latency of Real-Time Container Cloud for Software Radio Access Networks

Chen-Nien Mao, Mu-Han Huang, Satyajit Padhy, Shu-Ting Wang,
Wu-Chun Chung, Yeh-Ching Chung, and Cheng-Hsin Hsu
Department of Computer Science
National Tsing Hua University
Hsin Chu, Taiwan

***Abstract*—As the huge growth of mobile traffic amount, conventional Radio Access Networks (RANs) suffer from high capital and operating expenditures, especially when new cellular standards are deployed. Software, and cloud RANs have been proposed, but the stringent latency requirements e.g., 1 ms transmission time interval, dictated by cellular networks is difficult to satisfy. We first present a real software RAN testbed based on an opensource LTE implementation. We also investigate the issue of quality assurance when deploying such software RANs in cloud. In particular, running software RANs in cloud leads to high latency, which may violate the latency requirements. We empirically study the problem of minimizing computational and networking latencies in lightweight container cloud. Our experiment results show the feasibility of running software RANs in real-time container cloud. More specifically, a feasible solution to host software RANs in cloud is to adopt lightweight containers with real-time kernels and fast packet processing networking.**

***Index Terms*—Containers, RAN, Latency, Testbed**

I. Introduction

The global mobile traffic amount is expected to increase from 52 million terabytes in 2015 to 173 million terabytes in 2018 [1]. Such a huge growth imposes two major challenges on the existing Radio Access Networks (RANs). Next generation RANs have to: (i) offer higher bandwidth and (ii) consume less energy. The development of next generation RANs, as we witnessed, is an *evolutionary* process, which takes many iterations before mature. However, traditional RAN implementations and deployments are hardware-dependent, which lack for elasticity and incur high Capital Expenditure (CAPEX) and Operation Expenditure (OPEX). One way to overcome the limitations due to traditional, hardware, RANs is to implement *software* RANs, which relieve telecom service providers from repeatedly upgrading their equipment. It is well recognized that the software RAN is the enabler of next generation cellular networks [2].

While software RANs are more elastic, they are also more computationally intensive because most computations are done on *general purpose* processors. To be cost-effective, it has been purposed to deploy software RANs in cloud, in order to leverage its abundant and elastic resources. We refer to software RANs in cloud as cloud RANs throughout this paper. Realizing cloud RANs is no easy task, because of stringent latency requirements of cellular networks. For example, the LTE standards dictate a 1 ms transmission time interval [3]. To fulfill such a latency requirement, we have to carefully divide a cloud RAN into several software modules, and deploy individual modules at heterogeneous cloud servers. For example, delay sensitive processes are better put at edge cloud for lower latency, while computational heavy process are better put at data center for more horsepower. That is, *quality assurance* provided by cloud is crucial to the success of cloud RANs. In particular, both computational and networking latencies need to be *guaranteed* before the cloud RANs become a reality.

In this paper, we study the problem of minimizing computational and networking latencies among several virtualized servers, so as to provide quality assurance in latency. Computation and server virtualization can be done via full-fledged Virtual Machines (VMs) or lightweight containers. While VMs offer better protections, they also incur higher overhead and may not be suitable to cloud RANs. Therefore, we adopt containers and strive to minimize the latency by leveraging: (i) real-time Linux kernel, which is key for fine-grained switching among real-time applications (ii) Data Plane Development Kit (DPDK), which is a framework for fast packet processing in data plane applications [4]. More precisely, we build real testbeds to quantify the performance of several real-time Linux kernels and DPDK networking. Our extensive measurement results reveal that hosting cloud RANs in real-time containers cloud *satisfies* the latency requirements from next generation RANs. Furthermore, we also set up a complete software RAN using off-the-shelf hardware and opensource software. Our end-to-end experiments show promising performance results. Decomposing the software RAN into multiple modules and deploying them in several containers of our real-time container cloud is our on-going task.

II. Related Work and Background

A. Cloud RANs in the Literature

As the cellular network technology evolves from GPRS/EDGE, UMTS/HSPA, to LTE/LTE-A [5], base stations sustain far more traffic than before. Studies propose small cells [6] as a solution to rapid growing traffic coming from the huge numbers of mobile devices. However, small cells face various challenges on OPEX/CAPEX, interferences, and mobility managements [7]. Higher OPEX/CAPEX prevents telecoms from deploying small cells in larger scale.

978-1-4673-9560-1/15 $31.00 © 2015 IEEE
DOI 10.1109/CloudCom.2015.67

Cloud RANs aim for the balance between performance and expense. Gudipati et al. [8] proposed SoftRAN, a logical, software-defined centralized control plane for radio access networks. However, it does not performed centralized baseband processing in cloud. FluidNet [9], [10] is a cloud RAN prototype with a BaseBand Unit (BBU) pool partially supported by DSP processors. It determines configurations that maximize the traffic sustainability under real-time requirements, while optimizing the computing resource usage of BBU pool. Sora [11] and BigStation [12] demonstrate the feasibility of real-time processing of wireless networks, such as WiFi, on commodity servers. CloudIQ [13] is the first work to run LTE BBU on commodity servers with full-stack LTE software implementation. It partitions base station instances into modules, and schedules base station instances to meet their real-time requirements with real-world workloads. PRAN [14] presents a high-level architecture that puts L1/L2 processing of BBU pool onto commodity servers. It enables flexible data paths and efficient resource pooling. The aforementioned studies do not capitalize the elasticity and scalability of cloud, nor quantify the performance of cloud RAN on real infrastructure cloud, such as Kubernetes and Openstack.

B. Container-Based Virtualization

Container-based virtualization becomes more and more popular because of its low system overhead and short launch time compared to VMs. It is relatively lightweight since it includes only necessary system libraries and binaries instead of entire kernel. Container has demonstrated its strength on computing [15], storage [16], and network emulation [17].

Linux Container (LXC) is the most well known one among various container implementations. It provides an execution environment with isolated system resources and its own file system. Namespace is used for resource isolation, while cgroup is used for process control. Recent research [15], [18] conducts in-depth performance evaluation on Linpack, memory, disk I/O, and networking. LXC demonstrates much lower system overhead than other virtualization techniques.

Docker is a container-based platform which provides high-level APIs that allows user to build, ship and run applications in containers easily. Docker was released first based on LXC technique, then using its own libcontainer library for Linux kernel's virtualization since version 0.9. Compare to plain LXC, Docker accommodates high-level tools such as layered file system (AUFS) and image registry for developers. For application deployment, Docker provides a image registry for users to push their own instances. We can deploy our applications, pack the instances as a service and push to the repository. With Docker, we can build a simple virtualized environment, pull and launch instances on-demand readily. With the rapid growth of Docker, Google initiates Kubernetes to manage the containerized applications in a clustered environment.

Kubernetes provides basic mechanisms to facilitate service deployment and maintenance based on Docker containers. In Kubernetes system, master servers manage the workloads in the whole system, store configuration data of all instances, and track resource utilization on each node. Minion servers will run actual workloads, and are also responsible for network configurations among containers. Kubernetes has four units: Pod, Replication Controller, Service, and Label.

- **Pod**: Pod is the basic unit in Kubernetes environment. It may contain one or more containers which are tightly coupled. Containers in the same pod use the same network namespace, which facilitates data sharing and communication among each other.
- **Replication Controller**: Replication controller is responsible over maintaining a desired number of copies. It ensures that a specified number of replicas exist for service recovery, and provides scaling and rolling updates among replicas.
- **Service**: A service is an abstraction of real application service that defines a set of backend containers and the interface to access. Users can simply access from specific IP and port, and the requests and redirect to appropriate backends.
- **Label**: A label is a tag given by Kubernetes when an instance is created. This tag helps administrators to classify, organize, and monitor instances as a group.

III. Considered Cloud Radio Access Networks

A. Overview

We first deploy an opensource LTE implementation, OpenAirInterface (OAI) [19] on general purpose processors, which runs on Intel Linux. OAI is a developing project maintained by EURECOM and consists of full stacks of LTE and 3GPP standards. OAI implements the LTE architecture into two major modules: (i) Evolved Node B (ENB) as the base station and (ii) Evolved Packet Core (EPC), which is composed of Serving Gateway (S-GW) and Packet Data Network Gateway (P-GW) for data signal processing, Mobility Management Entity (MME) for control signal processing, and Home Subscriber Server (HSS) as the database. OAI needs an RF front end to receive or transmit the LTE signals. Both Universal

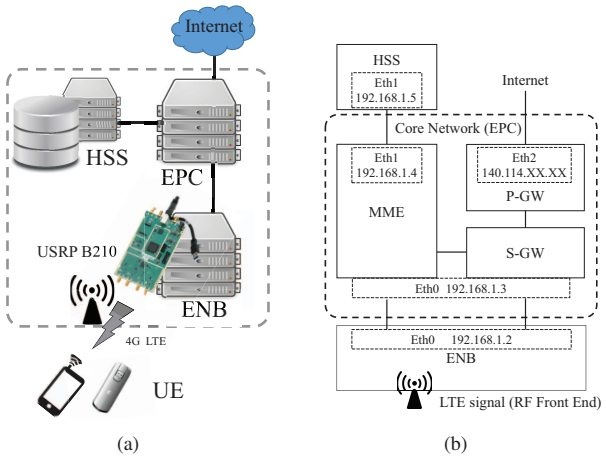


Fig. 1. Software RAN: (a) end-to-end architecture and (b) network topology.

Software Radio Peripheral (USRP) B210 USB board and ExpressMIMO2 PCI board made by EURECOM are compatible to OAI.

The features of OAI implementations are listed as follows:

- Implementation of L1/L2, Radio Link Control (RLC), Packet Data Convergence Protocol (PDCP) and Radio Resource Control (RRC).
- Protocol stacks from the physical layer to the networking layer.
- FDD and TDD modes supports.
- Built-in emulator and simulator.
- IoT supports, such as: Contention-based Channel Access (CBA) for M2M communications.

B. Testbed

Fig. 1(a) shows the architecture of our testbed. The OAI is deployed on the commodity PCs. A laptop computer equipped with an LTE dongle acts as the User Equipment (UE). We separate the OAI into ENB, EPC, and HSS components and deploy these components on different physical machines. Each physical machine comes with Intel i7-4790 CPU and 4 GB RAM. The kernel is 3.18.11 with real-time patch. We turn off the power saving mode and maximize the CPU frequency for further performance improvement. As shown in Fig. 1(b), the ENB, EPC and HSS components are connected via Ethernet and the RF front end is USRP B210. The UE is a Hauwei E3372 LTE dongle with a configurable SIM card for connecting to the LTE software modem.

Once UE sets up RRC connection with ENB, the authentication procedure is handled by MME and HSS. If the information of UE is well configured in the HSS, UE can attach to MME via the ENB unit. Then, the Evolved Packet System (EPS) bearer for the UE and data flow are established. In this setup, we have verified the connection of OAI and the authentication procedure on MME. After the success of authentication, the data transmission is established and the UE has access to the Internet via our testbed.

C. Limitations

Our testbed exploits a software RAN to set up a LTE base station and deploy a core network on the PCs with general purpose processors. We measure the CPU usage and the network latency on our testbed to quantify its performance. Preliminary results show that the CPU usage is 52% and the throughput is 0.8 Mbps on average when we run the real LTE soft modem to download a 100 MB file from a remote file server (25 resource blocks for downlink). When we change the OS to the real-time kernel, the CPU usage increases for 15%. Moreover, the ping latency to our network gateway is 32 ms on average which needs further improvement for the next generation RANs. We optimize the latency in the next section to cope with the limitations.

IV. Latency Optimization

We build a real-time cloud using Docker for containers, Kubernetes for containerized application management, and Intel's Data Plane Development Kit (DPDK) for low networking latency. We then optimize the container cloud in two aspects: computation and networking below.

A. Computations: Docker with Real-time Kernel

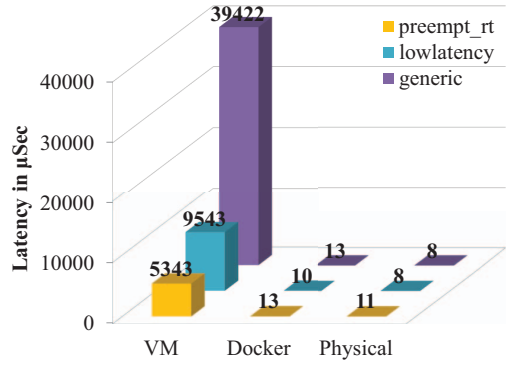
RANs require robust real-time processing capability, which are mostly handled by dedicated Digital Signal Processing (DSP) processors. Cloud RANs, however, rely on general purpose processors supported by real-time kernels. There are two types of real-time kernels: hard real-time kernel and soft real-time kernel. Hard real-time kernel guarantees that the requested critical tasks completed in time even under the worst system loads. The real-time guarantee comes from bounding all the delays in the system for the tasks. Soft real-time kernel gives a critical task higher priority than other tasks until it is completed. It reduces average latency but it may not be completed on time. We select two real-time kernels for our cloud RAN environment: preempt_rt kernel (preemptive real-time kernel for Linux) as a hard real-time kernel and low-latency kernel as soft real-time kernel.

Our preempt_rt kernel is built by applying Ingo Molnar's real-time preemption patch and tuning real-time kernel attributes related with lower latency. It makes the locking primitives in kernel preemptible with rtmutexes so that higher priority tasks can be preempted in the user space itself rather than waiting for the kernel space. It should be noted that the preempt_rt kernel is tuned with many real-time kernel attributes to provide low latency in the system. On the other hand, we apply the low-latency kernel, which is available on the Ubuntu repository. It achieves low latency in a different way compared to the preempt_rt kernel.

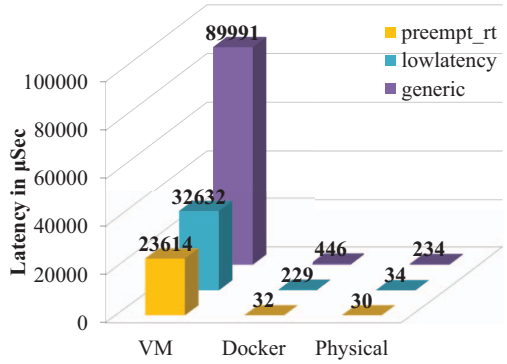
We use a SuperMicro server for the entire experiments. The server has 128GB RAM and 24 CPUs. Each CPU is an Intel Xeon CPU E5-2620 v3 with frequency 2.4 GHz. We have installed Ubuntu Server 14.04 with a stock kernel generic-3.13. Both preemptive-rt kernel and low latency kernel are installed on our server. Some kernel parameters are fine tuned in preempt_rt kernel to achieve lower latency. The experiments are performed on three different type of kernel: generic-3.18.11, preempt_rt-3.18.11, and low latency-3.18.11. The latencies are compared among the physical machines, containers, and VMs with libvirt manager.

Cylictest [20] benchmark is used to evaluate latency performance, such as maximum and average latencies, in our underlying system. We are interested in the maximum latency so that we can estimate the performance of the worse case. We choose First-In-First-Out among several scheduling policies in the kernel since it gives better latency than others. We have run 500,000 cycles of Cyclictest benchmark to evaluate our latency. The experiment is performed on two scenarios: one is without system load and another is with system load. We use *stress* [21] to generate workload by forking worker processes until the system reaches 100% utilization.

As discussed above about the experimental environment and the proposed scenarios, the results are the maximum latencies in microseconds that are compared between three different



(a)



(b)

Fig. 2. Cyclictest Benchmark with 500,000 cycles: (a) without and (b) with system load.

types of system on three types of kernels in Fig. 2. The observations are as stated below:

- Preempt_rt kernel improves latencies on Docker and physical machines under system loads with 13.9 times and 7.8 times compared to generic kernel, respectively. It also achieves the lowest latency than other kernels as shown in Fig. 2(b).
- Nevertheless, it fails to show its strength on real-time processing without system load, so very little improvement is shown in Fig. 2(a) for this circumstance eventually.
- Among all the systems, the physical machine helps in achieving the lowest latency which is generally expected and the docker containers give the second lowest latency.
- The latencies in VMs are more on the higher side. Comparing it to the other two systems with docker and physical machines, the latency in virtual machine is really high.

Furthermore, if we upgrade the hardware configurations of the experimental environment, there are some key observations made. Testing it with different environments, docker containers achieve shorter latency compared to VMs and physical machines.

There is a high demand of a real-time environment for cloud RANs and more often there will be high system load for heavy processing. After comparing and analyzing the latencies, preemptive_rt achieves the lowest latency of 34 microseconds under heavy system load.

It's highly demanding to tackle with real-time workloads in cloud RANs. These workload can be represented by the workload generated by stress. Therefore, we compare latencies between docker and physical machines which stands for native performance with fully occupied system load. The difference between Docker and physical machines is only 2 μs with preempt_rt kernel in Fig. 2(b). Consequently, we can conclude that Docker achieves near-native performance of 34 μs latency, while the latencies in virtual machine are way too high.

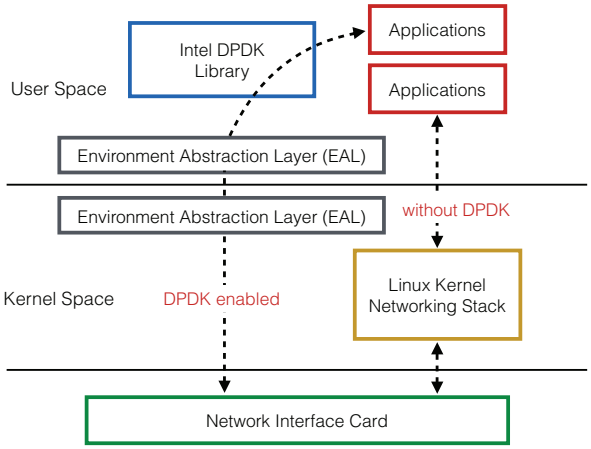


Fig. 3. Architecture of Intel Dataplane Development Kit (DPDK).

B. Networks: Kubernetes with DPDK

Packets are transferred to main memory from Network Interface Card (NIC), then subsequent processing is handled by Linux network stack. In general, network performance is mainly bounded by CPU-cycles, memory allocation, and number of context switches between user and kernel mode. Simultaneous hardware accesses on cloud infrastructure increase network latencies. To overcome the limitations of Linux network stack for faster packet processing, several techniques are proposed.

First, Netmap [22], an opensource project, successfully reduces three types of packet processing overhead: per-packet dynamic memory allocations, redundant system calls, and multi-memory copies. Moreover, Netmap provides its own software-based mechanism to protect shared memory from kernel crash. Second, PF_RING project [23] proposes a new type of network socket for faster packet capture under more efficient CPU usage. It uses Linux NAPI drivers to poll packets from NIC to PF_RING circular buffer. Userspace applications can read packets directly form the circular buffer. PF_RING provides useful features such as memory pre-allocation for packet buffers, parallel direct paths using memory mapping,

and the integration with PF_RING Direct NIC Access module [24], which directly maps NIC memories to userspace.

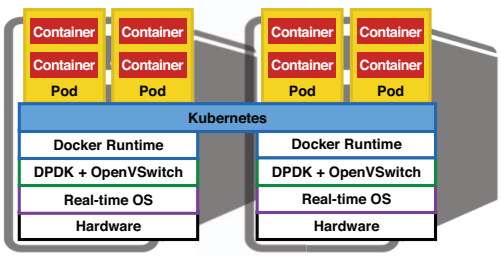


Fig. 4. Platform architecture: DPDK-enabled Kubernetes based on real-time kernel.

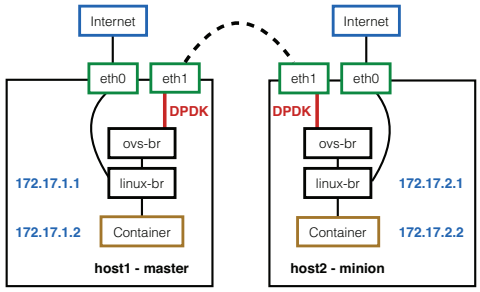
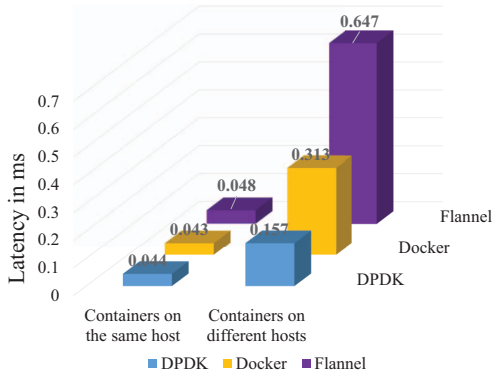


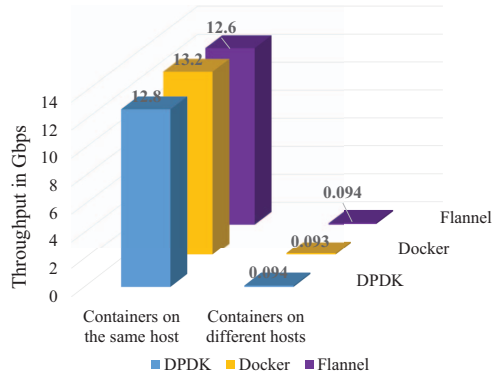
Fig. 5. DPDK-enabled OpenVSwitch network architecture in Kubernetes platform.

Third, Intel Data Plane Development Kit (DPDK) [4], [25] enables fast packet processing for data plane applications. In DPDK architecture shown in Fig. 3, several libraries are provided to carry basic functions of Linux network stack, and the optimization for memory/buffer allocation and mapping. DPDK-enabled OpenVSwitch supports network flow management and reduces network latency. By adapting DPDK in our platform, the massive network traffics can be handled through Environment Abstraction Layer to have fast access to hardware and memory. However, most of the fast packet processing frameworks provides very basic network flow management utility, slightly advanced functionalities, such as IP forwarding, are not supported. Openflow, the de facto standard of Software Define Networks (SDN), solves the network flow management issues with DPDK compatibility. Therefore, we chose DPDK-enabled OpenVSwitch in our testbed.

DPDK uses igb_uio module to bind a NIC, which makes the NIC invisible to Linux kernel. Therefore, we need NIC for Internet access and cross-node communication, respectively. We use another testbed for network latency experiments, which contains two T-win servers with Intel Xeon L5640 CPU and 24GB RAM. It runs 64-bits Ubuntu 14.04 with preemptive real-time kernel 3.18. Servers are equipped with DPDK-compatible NIC and they are connected via Ethernet. We install Kubernetes 1.0, OpenVSwitch 2.0.2, and DPDK



(a)



(b)

Fig. 6. Network performance in containerized environments over different network architectures: (a) latency and (b) throughput.

1.7 for our testbed, as shown in Fig. 4. As for networking mechanism, Kubernetes uses flannel, a generic overlay network, which will allocate a subnet for each host, encapsulate IP fragments in a UDP packet to traverse packet between hosts. We replaced flannel with DPDK-enabled OpenVSwitch bridges to attain better performance. The network architecture is shown in Fig. 5, DPDK is enabled between ovs-br and eth1 for acceleration while IP routing is simply achieved by routing tables. Kubernetes pods with single container are created to evaluate the networking performance of our platform.

To quantify the networking performance of our platform, Fig. 6 plots the performance of latency and throughput. Several observations can be made in this figure. For docker containers with plain OpenVSwitch bridge, network latency is about 0.043 ms for single machine and 0.313 ms for containers on different hosts. With Kubernetes involved using original flannel for networking, the latency is about 0.048 ms in single machine, and cross-host communication increased to 0.647 ms due to multiple-layered NATs. In our platform, DPDK-enabled OpenVSwitch is adopted in Kubernetes, which achieves the best performance of 0.044 ms for single-host and 0.157 ms for cross-host networking. Last, the network throughput is pretty consistent across all network architectures. In summary,

DPDK reduces latency by up to 2 times compared to plain OpenVSwitch architecture and the same throughput as other architectures. We can conclude that DPDK is indispensable for building a low network latency cloud platform.

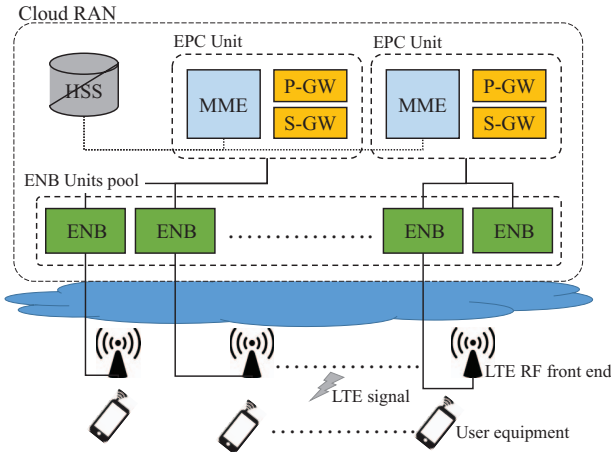


Fig. 7. Our work-in-progress cloud RAN deployment scenario.

V. Conclusion and Future Work

To satisfy the strict latency demand of running software RANs in cloud, we proposed two approaches to reduce the latency: (i) fine-tuned real-time kernel for processing latency optimization and (ii) docker with DPDK for networking latency optimization. The experiment results clearly demonstrate the effectiveness of our approaches to latency optimization. The resulting real-time container cloud is an *enabler* of various real-time Network Function Virtualization (NFV) services, including cloud RANs illustrated in Fig. 7. We plan to deploy each software component (of this figure) in a container for flexible resource provisioning and dynamic relocation. For a centralized solution [26], we will construct several ENB services to act as a resource pool managed by Kubernetes. If network congestion occurs, we can easily deploy more ENB services in the pool to provide quality assurance, and achieve high energy efficiency by dynamically allocating the resources. Furthermore, if an ENB fails, Kubernetes can recover the service via Replication Controller.

We presented a real-time container cloud for real-time NFV services. Several enhancements may be applied to our real-time container cloud for even lower latency. For example, we plan to implement deadline sensitive processes using OpenCL to leverage additional GPU horsepower. We also plan to integrate SDNs with our container cloud to dynamically allocate the resources of wired networks for end-to-end quality assurance. Last, splinting soft RAN into multiple components in distributed containers leads to additional overhead and low balancing concerns, we will need to be rigorously studied.

Acknowledgments

The authors would thank the anonymous reviewers for their insightful comments. This work was partially funded by Ministry of Science and Technology and Industrial Technology Research Institute, Taiwan, under grant numbers MOST104-2221- E-007-053 and ITRI104A0058SB, respectively.

References

[1] "Gartner gartner forecasts 59 percent mobile data growth worldwide in 2015," http://www.gartner.com/newsroom/id/3098617.
[2] C. I, C. Rowell, S. Han, Z. Xu, G. Li, and Z. Pan, "Toward green and soft: a 5G perspective," *IEEE Communications Magazine*, vol. 52, no. 2, 2014.
[3] "LTE-SAE architecture and performance," http://www.ericsson.com/ericsson/corpinfo/publications/review/2007_03/files/5_LTE_SAE.pdf.
[4] D. Scholz, "A look at Intels dataplane development kit," *Network*, vol. 115, 2014.
[5] S. Sesia, I. Toufik, and M. Baker, *LTE: the UMTS long term evolution*, 2nd ed. Wiley, 2011.
[6] V. Chandrasekhar, J. Andrews, and A. Gatherer, "Femtocell networks: a survey," *IEEE Communications Magazine*, vol. 46, no. 9, 2008.
[7] I. Hwang, B. Song, and S. Soliman, "A holistic view on hyper-dense heterogeneous and small cell networks," *IEEE Communications Magazine*, vol. 51, no. 6, 2013.
[8] A. Gudipati, D. Perry, L. Li, and S. Katti, "SoftRAN: Software defined radio access network," in *Proc. of ACM SIGCOMM HotSDN*, 2013.
[9] C. Liu, K. Sundaresan, M. Jiang, S. Rangarajan, and G. Chang, "The case for re-configurable backhaul in cloud-RAN based small cell networks," in *Proc. of IEEE INFOCOM*, 2013.
[10] K. Sundaresan, M. Arslan, S. Singh, S. Rangarajan, and S. Krishnamurthy, "FluidNet: a flexible cloud-based radio access network for small cells," in *Proc. of ACM MobiCom*, 2013.
[11] K. Tan, H. Liu, J. Zhang, Y. Zhang, J. Fang, and G. Voelker, "Sora: High-performance software radio using general-purpose multi-core processors," *Communications of the ACM*, vol. 54, no. 1, 2011.
[12] Q. Yang, X. Li, H. Yao, J. Fang, K. Tan, W. Hu, J. Zhang, and Y. Zhang, "BigStation: Enabling scalable real-time signal processingin large mu-mimo systems," in *Proc. of ACM SIGCOMM*, 2013.
[13] S. Bhaumik, S. Chandrabose, M. Jataprolu, G. Kumar, A. Muralidhar, P. Polakos, V. Srinivasan, and T. Woo, "CloudIQ: a framework for processing base stations in a data center," in *Proc. of ACM MobiCom*, 2012.
[14] W. Wu, L. Li, A. Panda, and S. Shenker, "PRAN: Programmable radio access networks," in *Proc. of ACM HotNets*, 2014.
[15] M. Xavier, M. Neves, F. Rossi, T. Ferreto, T. Lange, and C. Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Proc. of IEEE PDP*, 2013.
[16] J. Kang, B. Zhang, T. Wo, C. Hu, and J. Huai, "Multilanes: Providing virtualized storage for OS-level virtualization on many cores," in *Proc. of USENIX FAST*, 2014.
[17] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. N. McKeown, "Reproducible network experiments using container-based emulation," in *Proc. of ACM CoNEXT*, 2012.
[18] F. Wes, F. Alexandre, R. Ram, and R. Juan, "An updated performance comparison of virtual machines and Linux containers," *IBM Research Report*, vol. 28, 2014.
[19] N. Nikaein, M. Marina, S. Manickam, A. Dawson, R. Knopp, and C. Bonnet, "OpenAirInterface: A flexible platform for 5G research," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 5, 2014.
[20] "Cyclictest," http://step.polymtl.ca/~fgiraldeau/resources/papers/rtlws14_paper.pdf.
[21] "Stress web page," http://people.seas.harvard.edu/~apw/stress/.
[22] L. Rizzo, "netmap: A novel framework for fast packet I/O," in *Proc. of USENIX ATC*, 2012.
[23] "PF_RING web page," http://www.ntop.org/products/packet-capture/pf_ring/.
[24] "PF_RING DNA web page," http://www.ntop.org/products/pf_ring/dna/.
[25] "DPDK web page," http://dpdk.org/.
[26] "C-RAN: the road towards green RAN," http://labs.chinamobile.com/cran/wp-content/uploads/CRAN_white_paper_v2_5_EN.pdf, 2011.