Opportunistic CPU sharing in Mobile Edge Computing deploying the Cloud-RAN

Andres F. Ocampo*†, Mah-Rukh Fida‡, Juan F. Botero‖, Ahmed Elmokashfi§, Haakon Bryhni*
* SimulaMet – Simula Metropolitan Center for Digital Engineering, Oslo, Norway
† OsloMet – Oslo Metropolitan University, Oslo, Norway
‡ School of Computing and Engineering - University of Gloucestershir, Cheltenham, United Kingdom
§ Amazon Web Services (AWS), Seattle, Washington, United States
‖ Department of Electronics and Telecommunications Engineering, University of Antiouqia, Medellin, Colombia
Corresponding author: andres@simula.no

***Abstract*—Leveraging virtualization technology, Cloud-RAN deploys several virtual Base Band Units (vBBUs) along with collocated applications on the same Mobile Edge Computing (MEC) server. Nevertheless, sharing computing resources (e.g., CPU) with collocated workloads could impact the performance of vBBU and other instantiated real-time (RT) applications. To tackle such issues, this paper proposes a run-time/dynamic CPU sharing mechanism for containerized virtualization in MEC servers hosting RT applications such as the vBBU along with general-purpose applications. Formulating the CPU sharing problem as a mixed integer problem (MIP), the proposed mechanism is based on the decomposition of the MIP into simpler subproblems solved through efficient constant factor heuristics. We evaluate the algorithm's performance against optimal solvers. Then, using a small-scale testbed, we evaluate different CPU sharing mechanisms and their ability to mitigate the impact of CPU sharing on the processing performance of RT applications. This analysis is extended to the Cloud-RAN, where different CPU sharing strategies are evaluated showing how these mechanisms contribute to mitigating the impact on the vBBU performance. Our findings show that the proposed CPU sharing mechanism reduces the impact of computing resource sharing on vBBU scheduling latency, by up to 19%, compared to the default Linux RT-Kernel Scheduler.**

***Index Terms*—Cloud-RAN, Mobile Edge Computing, Containers, resource management**

I. Introduction

The unprecedented growth of wireless traffic demand and the time sensitive nature of services expected for the fifth generation of mobile systems 5G [1], pose stringent constraints of capacity and low latency over the mobile system. As a result, a paradigm shift on the design and architecture of the radio access network (RAN) is key to address these challenges. For instance, the traditional distributed RAN architecture is not economically feasible for dense deployments of small cells, which seems to be the most likely network scenario in 5G [2]. One reason is that small cells increase substantially the transmission capacity and, therefore, the peak data rate per cell.

To cope with such unprecedented wireless traffic demand with low latency capacity requirements, both the Cloud-RAN architecture and Mobile Edge Computing (MEC) provide a promising solution for the RAN design. Leveraging software defined wireless networking and virtualization technology [3], the Cloud-RAN deploys multiple vBBUs along with collocated services on the same MEC server. On the other hand, providing cloud computing capabilities at the very edge of the mobile network, MEC significantly reduces latency of mobile services while easing both processing and traffic pressure over the mobile system [4]. Furthermore, by sharing network and processing resources, the Cloud RAN architecture and MEC brings optimized operation and maintenance benefits to mobile network operators (MNO) and service providers.

Because the vBBU performs both time sensitive signal processing (e.g., layer 1 functions) and control functions that do not impose latency requirements, the vBBU runs a combination of real-time (RT) and non real-time (non-RT) processes on the MEC server. Similarly, MEC runs a wide spectrum of services with diverse requirements often imposing low-latency constraints. Hence, running on MEC, the vBBUs is expected to share resources with collocated RT and non-RT applications. Although the feasibility of running the vBBU on a MEC has been confirmed in related research [5], there is still a need to investigate the vBBU performance in the light of sharing computing resources, aiming at providing deterministic execution time. Similarly, although running RT applications using virtualization technology has been addressed in related research [6], [7], there remains the need of investigating RT processing in the light of sharing resources as in MEC.

To avoid processing interference with collocated processes [8], running RT processes on a set of isolated CPUs has been a common approach in RT systems [7], [9]. However, running an application on isolated CPUs increases CPU underutilization [10]. For instance, because of the bursty nature of mobile traffic, vBBUs processing often remains idle. Consequently, by running the vBBU on isolated CPUs, most of the CPU-time remains unused while computing off-peak traffic events [11]. To address this issue, this paper presents a CPU sharing mechanism for containerized virtualization instantiating either RT applications (RT containers) or non-RT applications (non-RT containers) in MEC. This mechanism, called PRINCIPIA, considers that RT-containers are allocated a fixed set (i.e., do not change over time) of CPUs at deployment. Then, leveraging Cgroups's [1] features to control resources assigned to containers, PRINCIPIA enables non-RT containers to exploit the underutilized CPUs allocated to RT-containers. First, by allocating non-RT containers on the same CPUs as the RT

[1] https://man7.org/linux/man-pages/man7/cgroups.7.html

containers, subject to CPU requirements. Then, by controlling the relative amount of CPU-time that non-RT containers are allowed to use on those CPUs.

After evaluating the performance and limits of PRINCIPIA when running a combination of RT and non-RT containers with different workloads, our study evaluates the benefits of adopting PRINCIPIA as a CPU sharing solution for containerized virtualization in MEC deploying the Cloud-RAN. To do so, this paper uses a small scale testbed deploying vBBUs with different functional splits, and switched Ethernet as mobile transport Xhaul network. The vBBUs are instantiated on a centralized MEC server along with collocated services with diverse processing latency requirements.

In summary, the main contributions of this paper are as follows.

- developing CPU sharing policies for containerized virtualization in MEC,
- evaluating the realization of RT services while sharing computing resources in MEC, and
- evaluating the benefits of adopting CPU sharing policies in MEC deploying the Cloud-RAN.

The rest of the paper is organized as follows: section II presents the background and motivation. Section III models the MEC server considered in this study, while section IV formulates the CPU sharing problem for collocated containers sharing CPU resources in MEC servers. This section also presents PRINCIPIA as a heuristic solution to the CPU sharing problem. In section V, we evaluate different CPU sharing approaches for collocated containers and their impact on the RT performance on RT applications. Section VI extends this evaluation to the Cloud-RAN, by assessing the impact that sharing computing resources with collocated applications causes on the RT processing performance of vBBUs, and how different CPU sharing mechanisms can be used to mitigate such impact. This section also assesses the impact of resource sharing on the mobile network end-to-end performance. Section VII presents the related work on Cloud-RAN architecture, and the RT support to run RT applications on containerized virtualization while sharing computing resources in MEC. Finally, section VIII concludes the paper.

II. Background and Motivation

This section covers the background by presenting the general components of the Cloud-RAN architecture, and discussing the deployment of vBBUs on top of MEC servers using containerized virtualization while sharing computing resources.

A. The Cloud-RAN architecture

The RAN consists of user equipment(s) (UE), the air interface, antennas, the Remote Radio Unit(s) (RRU), the Baseband Unit (BBU), and a network link connecting the RRU and the BBU known as Fronthaul. A transport network known as Backhaul connects the RAN to the Core Network (CN). As depicted in figure 1, in Cloud-RAN, the BBU is implemented as a software-defined wireless networking application (vBBU). Leveraging virtualization technologies, several vBBUs could be deployed on top of a centralized MEC server sharing processing and network resources [12].

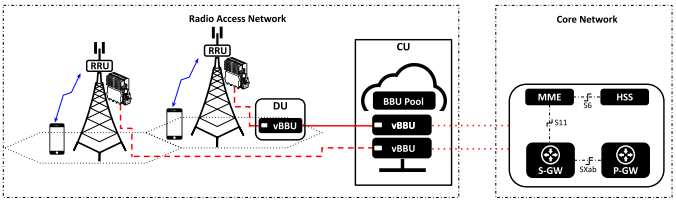


Fig. 1: Cloud Radio Access Network architecture.

By sharing computing and networking resources, the Cloud-RAN architecture provides a new paradigm to the RAN design. Nevertheless, centralizing the BBU poses stringent latency constraints and capacity requirements both to the Fronthaul network and to the MEC server hosting vBBUs. To tackle these challenges, two promising solutions are being considered and standardized.

Functional Split: Processing part of the BBU functions locally close to the antennas decreases the requirements of bandwidth and latency in the Fronthaul. For instance, the 3GPP proposes a functional split reference model based on the LTE protocol stack [13]. Moreover, the IEEE 1914 working group has defined two logical split points placement [14]: the distributed unit (DU), which is located near the cell tower; and the centralized unit (CU), which is located at the MNO's MEC. Introducing both split points redefines the mobile transport network segments and their requirements in terms of latency and capacity: the Fronthaul is the segment between the RRU and the DU; the Midhaul is the segment connecting the DU and the CU, where data rate requirements depends upon the chosen functional split; and the Backhaul, which connects the Cloud-RAN with the CN. These transport segments are referred to as mobile Crosshaul (XHaul).

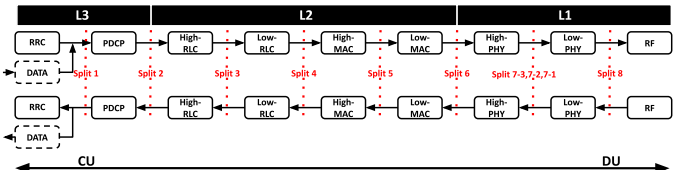


Fig. 2: 3GPP functional split of the LTE-BBU functionality [13].

Depicted in figure 2, the dotted red line highlights the split option as defined by 3GPP. Functions on the left side of a given option are instantiated at the CU, whereas right side functions are left for the DU. The more functions are instantiated at the DU, the less stringent latency and network capacity constraints over the Midhaul; the more functions are instantiated at the CU, the higher the requirements of capacity and latency on the Midhaul [15]. Figure 1 illustrates the functional split placement for two vBBUs. While one of the vBBUs centralizes all its functions (i.e., split 8), the second one implements functional splitting between the DU and the CU. Related works that study dynamic functional split and placement include the work in [16].

Switched Ethernet Mobile X-Haul network: Switched Ethernet statistical multiplexing enables a shared transport network, aggregating traffic flows from multiple BBUs into the same

multihop network, instead of deploying a dedicated fiber solution per BBU [17]. The survey paper by Gomes et al. [18] summarizes the benefits, challenges, and the current approaches addressing the issues that Ethernet brings as mobile Xhaul.

B. Running the vBBU on MEC

MEC servers run multiple applications with diverse execution time requirements [19], including RT applications such as the vBBU. To run RT applications on top of a MEC system, the host OS must provide RT guarantees, i.e., preemption and a scheduling policy that focuses on meeting timing constraints of individual processes rather than maximizing the average amount of scheduled processes. The incurred cost of development, maintenance, and licensing of a RT OS, has motivated the adoption of a general purpose OS like Linux to run RT systems [20]. Linux open source licensing and Kernel modularity, ease the development, customization and maintenance of any feature into the source code, thus reducing costs [21]. Moreover, several mechanisms have been proposed in recent years to provide RT support in the Linux Kernel (e.g, RTLinux [22], Low-Latency patch [23], PREEMPT_RT [24] patch), opening up the possibility of its use for RT systems [25], particularly, for RT signal processing of vBBU functions in [26].

Virtualization technology with RT support for running RT applications: Virtualization technology (e.g., hypervisor, containers) enables multiple applications running as isolated processes on the same computing infrastructure. Recent efforts to running RT applications using virtualization has motivated the adoption of the concepts RT VMs and RT containers, which refer to VMs and containers providing RT support, respectively [25], [27]. To provide RT guarantees in hypervisor based virtualization, while the hypervisor defines the RT scheduling mechanism that allocates CPU time to virtual machines (VM), the guest OS deploys an RT Kernel that preempts non-RT task on behalf of RT ones [28] (e.g., Linux with RT support). In containerized virtualization, on the other hand, containers do not deploy such guest OS. Instead, containers rely on Kernel's features Cgroups and namespaces to isolate processes [29]. Hence, to provide RT guarantees, containers rely on the host OS which adopts a RT-Kernel supporting preemption and RT Scheduling [6].

When running RT processes, containers provide better RT performance than VMs [25]. The reason has to do with the overhead generated by Hypervisor and the guest OS. Conversely, containerized virtualization is considered a lightweight virtualization as containers are isolated using Kernel features namespaces and Cgroups [30].

C. Sharing computing resources in MEC servers

While running RT processes on a MEC system managed by the Linux RT-Kernel, RT processes likely share computing resources either with collocated user-space processes or with Kernel threads. As a result, collocated workloads potentially induce processing interference, either from sharing physical resources [31], [32] or from Kernel space processing [33], [34] that could impact the performance of RT applications [35]. Despite the research efforts to understand the impact of resource sharing among mixed time-critically applications on general purpose processors (GPP), there remains the need to investigate such impact on time-sensitive applications deployed using virtualization technology, particularly in containerized virtualization. Moreover, there is a need for resource management solutions that enables resource sharing among collated applications while providing RT guarantees (i.e., enhanced determinism) to time-sensitive applications in MEC [36].

Runtime resource management provides an alternative solution for resource contention among collocated applications [37]. Typically deployed either at the Kernel level (e.g., task scheduling solutions [38]), or at the user-space (i.e., running as a daemon process [39]), runtime resource management allows efficient resource utilization while providing differentiated processing guarantees to heterogeneous applications [40]–[42]. In the context of MEC and multi-Cloud Computing in general, resource management caters a wide spectrum of problems [43], e.g., resource provisioning, resource allocation, resource mapping, service migration, among others. Typically implemented on resource orchestration platforms, solutions to these problems aim at providing dynamic multi-server/multi-Cloud resource management that optimize the utilization of physical resources among applications instantiated using virtualization technology [44]. Resource management solutions in MEC/Multi-Cloud computing considering time-critical services commonly rely on dynamic/static resource provisioning of RT VMs/containers [45]. Nevertheless, the inherent migration cost (in terms of time) of dynamically provisioning applications among servers could impact the RT performance of time-sensitive services [46].

To avoid performance degradation of RT services due to resource provisioning among MEC servers, this paper assumes that RT applications (e.g., vBBU) are provisioned on a single MEC server at deployment time and can not be migrated among servers afterwards. This approach is based on the evidence from embedded computing platforms running multiple applications with diverse execution time requirements [42]. On the other hand, to avoid processing interference from collocated RT applications, this paper assumes that containers hosting RT applications are allocated a set of orthogonal CPUs (i.e., RT applications can not be mapped on the same CPUs). Nevertheless, running applications on isolating CPUs increases CPU underutilization [10]. This paper proposes a dynamic CPU sharing mechanism for containerized virtualization that enables containers hosting non-RT applications to opportunistically share the CPUs assigned to RT containers.

Aiming at mitigating the impact that resource sharing cause on RT applications, the CPU sharing mechanism allocates CPUs and controls the amount of CPU-time that containers hosting non-RT applications are allowed to use on the CPUs assigned to containers hosting RT applications. To do so, this mechanism leverages **Cgroups** subsystems **cpuset** [2] and **CPU-**

[2]https://www.kernel.org/doc/Documentation/cgroup-v1/cpusets.txt

shares [3]. While the former allows defining the set of CPUs that a container is allowed to use when running its applications, the later defines the relative amount of CPU-time that the container is allowed to use on the assigned CPUs. Furthermore, our mechanism provides differentiated CPU-allocation to non-RT containers by defining priority policies.

III. MEC System model

This system model considers a MEC server consisting of M CPUs. Because this paper focuses on sharing and allocating CPUs among collocated containers, other physical resources such as memory or disk are considered part of the common resource pool, hence they are not part of this model. Sharing and allocating such resources are decisions taken by the host Kernel subject to specified orchestration constraints.

Using containerized virtualization, this system hosts a combination of applications with diverse execution time requirements, which are classified into three groups: RT applications; prioritized (PR) non-RT applications, which requires prioritized access to resources yet running as non-RT; best effort (BE) non-RT processes, which are default general purpose applications. Also, we assume that a container can only instantiate applications with the same execution time requirement. Consequently, containers are classified as RT containers, PR containers, and BE containers, as defined by the following indicator variable:

$$k = \begin{cases} 1 & \text{if the container instantiates RT applications} \\ 2 & \text{if the container instantiates PR applications} \\ 3 & \text{if the container instantiates BE applications} \end{cases}$$

Note that the decision of instantiating and provisioning applications into this MEC server is taken by the resource orchestration platform managing the MEC. In this model, we assume that RT containers are statically instantiated on this system during orchestration time, while non-RT containers (e.g., PR and BE) can be instantiated dynamically during runtime.

Let $L_1 = \{11, 12, \ldots, 1R\}$ represent the set of deployed RT containers. Similarly, let $L_2 = \{21, 22, \ldots, 2P\}$ and $L_3 = \{31, 32, \ldots, 3B\}$ represent the set of deployed PR and BE containers, respectively. While the set of all deployed containers is represented by $L = L_2 \cup L_2 \cup L_3$, the set of non-RT containers is represented by $L_p = L_2 \cup L_3$. During deployment, each RT container $r \in L_1$ is pre-allocated a set of orthogonal CPUs according to its CPU requirement $C_r \leq M$ as defined by the orchestration platform. The pre-allocated CPUs to each RT container is described by the indicator vector $\boldsymbol{I_r} = (I_{r1}, I_{r2}, \ldots, I_{rM})$, where I_{rm} is defined by:

$$I_{rm} = \begin{cases} 1 & \text{if container } r \text{ is allocated CPU } m \\ 0 & \text{otherwise,} \end{cases}$$

for all $r \in L_1$ and $m \in \{1, 2, ..., M\}$. As stated, the number of CPUs allocated to RT containers must satisfy their CPU requirements, so that $\sum_{m=1}^{M} I_{rm} = C_r$. By orthogonal allocation we referred to the fact that two RT containers can not be allocated the same set of CPUs, in other words the inner product $\langle \boldsymbol{I_i}, \boldsymbol{I_j} \rangle = 0$, for all $(i, j) \in L_1$ where $i \neq j$.

Conversely, non-RT containers (both PR and BE) do not have any such pre-allocated CPUs. They opportunistically try to use the underutilized CPUs assigned to RT containers. Based on the container's CPU usage and the per processor CPU utilization from CPUs assigned to RT containers, this model considers dynamic CPU allocation to non-RT containers. While CPU allocation decisions can change over time, such decisions should meet the CPU requirement $C_n \leq M$ for each non-RT container $n \in L_p$ as defined by the orchestration platform. Note that the CPU allocation here refers to deciding the set of CPUs that a given container is allowed to use. The host RT-Kernel schedules CPU-time to the container from the allocated CPUs, once available, according to the Kernel scheduling policy and the priority of the container's instantiated application.

To compute parameters, our model assumes a slotted time $t \in \{0, 1, 2, \ldots\}$. The timeslot here, though, differs from the MEC's cycle duration in that this timeslot defines the granularity or interval duration at which this model computes its parameters. In essence, this model relies on monitoring system metrics like per processor CPU utilization and container's CPU usage to make CPU allocation decisions to containers every timeslot. Because the timeslot granularity is arbitrarily defined, measuring system metrics samples (e.g., CPU utilization) likely capture instant or temporal spikes that potentially lead to wrong model computation. For that reason, this model first monitors systems metrics, and then computes model parameters based on both the current sample and the trend from previous monitored data. To do so, this model tracks these system's metrics through the introduction of virtual ring buffers. Such buffers are virtual in that they are maintained purely in software, while saving metrics statistics collected over the last W timeslots.

As stated, temporary spikes or drops in measured metric samples would affect the further model computation. For this reason, we compute model parameters (e.g., per-processor CPU utilization and container CPU usage) using a variation of the standard exponential moving average (EMA) [47]. Rather than computing such parameters based on the snapshot of the last sample, the modified EMA (mEMA) provides smooth predicted samples for a general time series by combining the Simple Moving Average (SMA) over a sliding window and an EMA. To illustrate how this model computes mEMA, consider a timeslot of arbitrary granularity and a time-slicing window of size $W_{[\text{timeslots}]}$. Also, consider the so-called ring buffer $X(t) = \{x(t-1), x(t-2), \ldots, x(t-W)\}$ which contains the previous samples of a general time series measured over the last W timeslots. First, compute the Simple Moving Average (SMA) of $X(t)$ given by $SMA(X(t)) = \sum_{w=1}^{W} x(t-w)/W$. Then, for a new data sample $x(t)$ compute the MEMA as follows:

$$mEMA(x(t)|X(t)) = \alpha x(t) + (1 - \alpha) SMA(X(t)) \quad (1)$$

[3] https://man7.org/linux/man-pages/man7/cgroups.7.html

where $\alpha = \frac{2}{W+1}$ is the smoothing constant which determines how fast the exponential weights decline over the past consecutive W periods [48]. While there are no specific $alpha$ values that are universally applicable or commonly used across different data, choosing the α value depends on the desired trade-off between responsiveness to recent changes and the smoothness of predictions [49]. While a smaller value of alpha gives more weight to historical data, a larger value of alpha gives more weight to the recent sample [50]. Therefore, to make sample predictions less sensitive to short-term fluctuations (i.e., instant spikes or drops in a measured metric sample), our model adopts a slicing window $W = 10$ timeslots (i.e., $\alpha = 0.18$).

Unlike common EMA implementations which keep the last predicted value as the contribution from previous data on the subsequent computation of new predictions, this approach uses the last measured samples in $X(t)$ as the contribution from previous data. Specifically, the SMA equally weighs the samples on $X(t)$ to compute the predicted sample for the current timeslot. Hence, the mEMA provides a predicted sample every timeslot that exponentially weights the average contribution of the last measured samples through the sliding window. The value obtained after the mEMA calculation is further used to compute this model's parameters. By the end of timeslot t, the ring buffer $X(t)$ is updated according to the current sample $x(t)$ and the slicing window W.

A. Computing per processor CPU utilization

The per processor CPU utilization refers to the ratio between the number of cycles (i.e., CPU-time) that a given CPU spent actually processing system workloads over the total amount of cycles in a timeslot [51]. Measuring the CPU utilization every timeslot, this model buffers the previous W CPU utilization measurements.

For each CPU $m \in \{1, 2, \ldots, M\}$, we define the virtual ring buffer $U_m(t)$ containing the previous W samples of CPU utilization. Updated every timeslot, this virtual buffer evolves according to the slicing window W as $U_m(t) = \{u_m(t-1), u_m(t-2), \ldots, u_m(t-W)\}$. Here, $u_m(t-w)$ represents the CPU utilization as measured in timeslot $(t-w)$, where $w \in \{1, 2, \ldots, W\}$.

As a function of a new sample $u_m(t)$ and the ring buffer $U_m(t)$, this model computes the CPU utilization for each CPU $m \in \{1, 2, \ldots, M\}$ following the EMA in (1), as follows:

$$U_m^*(t) = EMA\{u_m(t)|U_m(t)\} \tag{2}$$

The CPU utilization allows deriving the CPU availability, which provides a notion of the unused CPU-time on CPU m. Let $G_m(t)$ denote the CPU availability on CPU m in timeslot t, given by:

$$G_m(t) = 1 - U_m^*(t) \tag{3}$$

B. Computing container's CPU usage

The container's CPU usage refers to the ratio between the total amount of CPU-time used by the container to run its instantiated application, over the total amount of CPU cycles. To monitor the container's CPU usage, for each container $n \in L$, we define the virtual ring buffer $Q_n(t)$. Updated every timeslot, this virtual buffer evolves according to the slicing window W such that $Q_n(t) = \{q_n(t-1), q_n(t-2), \ldots, q_n(t-W)\}$. Here, $q_n(t-w)$ represents the CPU usage of container n as measured in timeslot $(t-w)$, where $w \in \{1, 2, \ldots, W\}$.

Denoted by $Q_n^*(t)$, this model computes the CPU usage of container n in timeslot t, following the EMA in (1) as a function of the new sample $q_n(t)$ and the previous W samples stored in the ring buffer $Q_n(t)$, as follows:

$$Q_n^*(t) = EMA\{q_n(t)|Q_n(t)\} \tag{4}$$

IV. Opportunistic CPU sharing

This section presents an opportunistic CPU sharing mechanism that seeks to provide non-RT containers with access to unused CPU resources while ensuring that the performance requirements of RT containers are not compromised. The proposed mechanism leverages Cgroups' features to control resources assigned to containers, using **cpuset** and **shares** subsystems.

A. CPU and CPU-shares Allocation problem - C2SAP

Consider a system controller that monitors and computes the per CPU availability $G_m(t)$ following (3), for each CPU $m \in \{1, 2, \ldots, M\}$. Similarly, the system controller monitors and computes the container's CPU usage $Q_n^*(t)$ following (4), for each deployed RT and non-RT container $n \in L$. According to the current CPU availability and the container's CPU usage, the network controller decides the CPU allocation to each non-RT container $n \in L_p$ subject to its CPU requirement C_n, every timeslot t. Let $\chi_n(t) = (\chi_{n1}(t), \chi_{n2}(t), \ldots, \chi_{nM}(t))$ represent the CPU allocation vector for the non-RT container n. Here, $\chi_{nm}(t) \in \{0, 1\}$ denotes the CPU m allocation decision, such that $\chi_{nm}(t) = 1$ if the CPU m is allocated to container n, and $\chi_{nm}(t) = 0$ otherwise. Let $\Omega_n(t) = \{\omega_{n1}(t), \omega_{n2}(t), ..., \omega_{nM}(t)\}$ be a collection of positive weights for the non-RT container n, where $\omega_{nm}(t)$ denotes the weight (i.e., the value) of allocating CPU m to the non-RT container n.

To avoid potential impact on RT containers caused by processing interference from collocated non-RT containers, the system controller allocates the relative amount of CPU-time (referred to here as CPU-shares) that each container is allowed to use on the allocated CPUs. Let $S_n(t) = (S_{n1}(t), S_{n2}(t), \ldots, S_{nM}(t))$ represents the CPU-shares decision vector for container $n \in L$, where $S_{nm}(t)$ denotes the CPU-shares allocation decision for container n when using CPU m. Let $\alpha_n(t)$ be a positive weight for container $n \in L$, which denotes the container's CPU demand on the CPU-shares allocation.

The objective is to design a CPU sharing policy that yields a CPU and CPU-shares allocation to all containers by solving the following optimization problem:

$$\max \quad \sum_{n\in L_p}\sum_{m\in M}\omega_{nm}(t)\chi_{nm}(t)+\sum_{n\in L}\sum_{m\in M}\alpha_n(t)S_{nm}(t) \tag{5}$$

$$\text{s.t.} \quad \sum_{m\in M}\chi_{nm}(t)\geq C_n, \forall n\in L_p \tag{6}$$

$$\sum_{n\in L_p}S_{nm}(t)\chi_{nm}(t)+\sum_{r\in L_1}S_{rm}(t)I_{rm}\leq 1, \forall m \tag{7}$$

$$\chi_{nm}(t)\in\{0,1\}, \forall n,m \tag{8}$$

$$0\leq S_{nm}\leq 1, \forall n,m \tag{9}$$

Here, while the objective function (5) is a weighted sum of both binary and continuous decision variables, inequality (6) represents the constraint imposed by the CPU requirement of the non-RT container n. Inequality (7) represents the constraint imposed by the total CPU-shares containers (both non-RT and RT) sharing the same CPU. The above problem is similar to the variant of the mixed integer setup knapsack problem (MISKP) presented by Altay et al. in [52] where fractions (i.e., continuous decision variables) of individual items (i.e., integer decision variables) are allowed to be loaded into a capacitated knapsack. Here, the continuous decision variables $S_{nm}(t)$ represent the CPU-shares that allocate the relative amount (i.e., fraction) of CPU-time that containers are allowed to use on the allocated CPUs. Nonetheless, unlike the MISKP where the integer variables represent the number of items from different classes to be loaded in a common capacitated knapsac, the CPU allocation represented by the binary decision variables $\chi_{nm}(t)$ reduced to a generalized Maximum Weight Matching problem in a bipartite graph (as shown in figure 3), where the containers n are matched (i.e., assigned) to the CPU m based on their respective demands and available resources.

Involving both integer and continuous decision variables, mixed integer programming (MIP) is hard to solve in general (i.e., NP-hard). However, as highlighted by Andrea Lodi in [53], 50 years of Integer and MIP has led to a stable algorithmic approach for solving MIP problems efficiently in practice. Such an algorithmic approach relies on the interactive solution of Linear Programming (LP) relaxation of the original MIP problem and employing various branching heuristics, cutting planes, and primal-dual methods to derive optimal or close-to-optimal. For instance, for the MISKP [52], Altay et al. adopted the notion of Benders decomposition [54] to iterative fix the values of the binary decision variables generating a continuous LP-relaxation of the problem which is easier to solve. Similarly, we adopt the notion of Benders decomposition to solve the CPU and CPU-shares problem, which includes a subproblem containing only binary variables and an LP subproblem containing only continuous variables.

B. C2SAP decomposition

The C2SAP is decomposed into two parts: the CPU allocation problem, which contains only binary decision variables, and the CPU-shares problem, which contains continuous decision variables. The solution of the CPU allocation problem allows for fixing the binary variables to produce an LP for the CPU-shares problem.

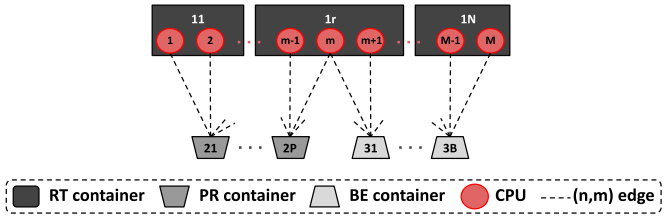


Fig. 3: Example MEC system showing containers with different priority (e.g., RT, PR, and BE). RT containers are pre-allocated a subset of CPUs. For each non-RT containers and each CPU there exist an edge (n,m) whose weight defines the basis for opportunistic CPU allocation.

1) CPU Allocation problem: The CPU allocation problem is obtained by skipping the CPU-shares part from the C2SAP. Furthermore, without losing generality, we assume that each non-RT container is allocated exactly the amount of CPUs as its CPU requirement C_n, avoiding overprovisioning of non-RT containers. To do so, rather than tightening the constraint (6) (i.e., changing the inequality constraint to an equality constraint), we redefine the decision variable to include the CPU requirements of each non-RT container. Let $c_n = \{ni | n \in L_p, i = \{1, ..., C_n\}\}$ be the set of CPU requirements of non-RT container n. Also, let $\chi_{n'm}(t)$ represent the decision of allocating CPU m to the CPU requirement n' of the non-RT container n in timeslot t, with $n' \in c_n$.

Choosing a CPU allocation $\chi_{n'm}(t)$ requires solving a generalized Maximum Weight Match problem (MWMP) on a $\sum_{n\in L'_p}|c_n| \times M$ bipartite graph, as illustrated in figure 3. Let $G(V, E)$ denote the bipartite graph where the vertex set V is decoupled into the set of CPU requirements of non-RT containers $V_1 = \bigcup_{n\in L_p} c_n$ and the set of CPUs $V_2 = \{1, 2, ..., M\}$, such that $V = V_1 \cup V_2$, and $V_1 \cap V_2 = \varnothing$. For each pair of vertices $n' \in V_1$ and $m \in V_2$, there exists an edge $(n', m) \in E$ whose weight is given by $\omega_{nm}(t)$ which depends upon the CPU availability and CPU usage of the non-RT container n. Thus, the CPU allocation decision consist of finding a matching $\chi_{n'm}(t)$ of maximum weight in the following problem:

$$\max \quad \sum_{n',m}\omega_{nm}(t)\chi_{n'm}(t) \tag{10}$$

$$\text{s.t.} \quad \sum_{n'\in V_1}\chi_{n'm}(t)=1, \forall m \tag{11}$$

$$\sum_{m\in V_2}\chi_{n'm}(t)=1, \forall n'. \tag{12}$$

$$\chi_{n'm}(t)\in\{0,1\}, \forall n', m \tag{13}$$

meaning that a vertex in a matching can not be adjacent to more than one vertex [55]. With this condition, though, once a CPU is allocated to a non-RT container (i.e., an edge is matched $\chi_{n'm}(t) = 1$), the same CPU can not be allocated to any other container, which is against the CPU sharing goal of this paper. Furthermore, if $|V_1| > |V_2|$, these

constraints violate the CPU requirement constraint (6) of the C2SAP. Allowing CPU sharing in the CPU allocation problem requires solving the MWPMP (which stands for Maximum Weight Perfect Matching Problem) in an augmented complete graph $G'(V', E')$. This augmented graph is constructed by adding dummy nodes to the vertex set $V_2{}' = \{mn'|\forall m \in \{1, 2, ..., M\} \wedge n' \in c_n\}$, such that $V' = V_1 \cup V_2{}'$. Also, we assume that G' is complete, meaning that for each pair of vertices $n' \in V_1$ and $m \in V_2{}'$, there exists an edge $(n', m') \in E'$ whose weight is given by $\omega_{n'm'}(t) = \omega_{nm}(t)$ if $n' \in c_n$, $\omega_{n'm'}(t) = 0$ otherwise. Without losing generality, the MWPMP reduces to a MWMP in an augmented graph by adding zero-weight edges when necessary [56]. Thus, solving the MWPMP on the augmented graph is equivalent to solving the MWMP on the original graph G.

The MWMP in bipartite graphs is a well-known problem in combinatorial optimization used to model a wide range of assignment problems [57]. Several algorithmic approaches have been proposed to find the exact solution to the MWMP, including the classical Hungarian [58] and Blossom [59] algorithms. Duan et.al in [60], [61] present a literature review of the most relevant algorithms to solve both the MWMP and the MWPMP while proposing linear-time approximation solution to the problem. Here, we use the Blossom algorithm as exact solution to the above problem establishing a baseline for comparison with a solution based on constant factor approximation discussed in the following.

2) CPU-shares Allocation problem: The CPU-shares Allocation problem is obtained by fixing the values of the binary decision variables $\chi_{n'm}(t)$ in the C2SAP, as the solution of the CPU allocation problem above. While fixing the $\chi_{n'm}(t)$ values converts the CPU allocation part of the objective function (5) into a constant, the integrality constraint (8) converts into the upper bound for the CPU-shares $S_{n'm}(t)$ decision variables. Thus, choose a CPU-shares allocation $S_{n'm}(t)$ that solves the following problem:

$$\max \quad \sum_{n\in L}\sum_{m\in M}\alpha_{nm}(t)S_{n'm}(t) + \sum_{n',m}\omega_{nm}(t)\chi_{n'm}(t) \tag{14}$$

$$\text{s.t.} \quad \sum_{n\in L_p}\sum_{n'\in c_n} S_{n'm}(t) + \sum_{r\in L_1} S_{rm}(t)I_{rm} \leq 1, \forall m \tag{15}$$

$$S_{n'm}(t) \leq \chi_{n'm}(t), \forall n, m \tag{16}$$

The above is a linear programming (LP) that seeks to maximize a weighted sum of CPU-shares. Here, inequality (15) represents the constraint imposed by the sum of CPU-shares allocated to containers sharing the same CPU, including the RT-containers provisioned to that CPU, which can not exceed 1. Inequality 16) represents the CPU-shares upper bounds constraint imposed by the CPU allocation such that the CPU-shares is zero if a given CPU is not allocated to a container in timeslot t.

Although the above C2SAP decomposition breaks down the problem into easier to solve CPU allocation and CPU-shares allocation subproblems, the optimal solution to these subproblems requires two different algorithms incurring significant overhead. For instance, solving the MWPM in an augmented graph involves $\sum_{n\in L_p'} |c_n|$ dummy nodes increasing the problem size and hence the computational complexity. To address these challenges, we next present a constant factor approximation algorithm for the CPU and CPU-shares allocation.

C. PRINCIPIA: opportunistic CPU sharing

Presented here is a constant factor approximations algorithm to the C2SAP presented earlier. While jointly solving the CPU and CPU-shares allocation subproblems, this algorithm is simpler to implement and results in lower computational overhead compared to exact methods.

To compute the weights on the CPU allocation, inspired by the Law of Attraction, the weight $\omega_{nm}(t)$ is defined as the attraction of a non-RT container n to use the CPU m as follows:

$$\omega_{nm}(t) = \frac{G_m(t)Q_n^*(t)}{\rho_m(t)^2} \tag{17}$$

Here, while the per processor CPU availability $G_m(t)$ represents the "mass" of CPU m, the container's CPU usage $Q_n^*(t)$ represents the "mass" of the non-RT container n. Similarly, representing the "distance" between the CPU m and a given container n, $\rho_m(t)$ computes the number of containers for which CPU m has been allocated until the beginning of timeslot t, with $\rho_m(t) \geq 1$ over slots $t \in \{0, 1, 2...\}$. The minimum value of $\rho_m(t) = 1$ is the baseline value indicating that only the RT container pre-assigned to CPU m is running and any additional non-RT containers have yet to be allocated CPU m until timeslot t. Without loss of generality, the proposed mechanism can also include allocating CPUs that have not been pre-assigned to any RT container. In such a case, the baseline value $\rho_m(t) = 1$ indicates that only the host Kernel is running on CPU m until timeslot t.

Furthermore, inspired by the common Inverse-Square Law, the weight of container n on the CPU-shares allocation $\alpha_n(t)$ is computed as follows:

$$\alpha_n(t) = \frac{Q_n^*(t)}{\upsilon_k^2} \tag{18}$$

Here, υ_k denote the control constant associated with each priority policy $k \in \{1, 2, 3\}$, such that $\upsilon_1 < \upsilon_2 < \upsilon_3$. Thus, the weight $\alpha_n(t)$ varies proportionally to the container's CPU usage $Q_n^*(t)$, and varies inversely as the square of the control constant υ_k. Put another way, the CPU intensity of container n decreases as its control policy is not RT (i.e., $k = 1$). For example, if the control constant of a PR container is twice that of RT containers (i.e., $\nu_2 = 2\nu_1$), it makes the intensity or weight of PR containers to be four times weaker than that of RT containers. Hence, υ_k enables controlling the influence of CPU intensive non-RT containers on the allocation of CPU-time. Although RT containers running RT applications can preempt non-RT containers in the RT-Kernel, limiting the proportion of CPU-time that collocated non-RT containers are allowed to use would benefit RT containers by reducing processing interference.

PRINCIPIA CPU and CPU-shares solution: PRINCIPIA is an approximate algorithm that jointly solves the CPU and CPU-shares allocation problems following the C2SAP decomposition introduced earlier. Rather than solving the MWPMP in an augmented graph, PRINCIPIA proposes a Greedy Maximal Weight Match (PGMWM) solution for the CPU allocation subproblem. The PGMWM aims to provide an online solution that can be computed with less overhead than the MWPMP which can be solved in polynomial time [62]. For instance, the Blossom algorithm requires up to $O(|V|^3)$ time complexity for solving the MWPMP [63], yet such complexity is often not feasible to implement in most practical scenarios [64].

Simpler greedy maximal matching (GMM) algorithms have been widely adopted in practice which significantly reduces the algorithmic complexity [64]. The most common performance metric is the worst-case ratio between the size of the matching obtained by the algorithm and the size of the maximum matching. Reported randomized greedy algorithms have achieved worst-case ratios above 0.7 [65]. To meet each container's CPU requirement, the PGMWM provides an iterative solution to the CPU allocation problem that achieves a worst-case ratio of 1 (perfect matching). The basic idea behind PGMWM is to select edges in decreasing order of weight. Similar GMM algorithms have been developed for general randomized graphs [66]. More specifically, the PGMWM is similar to the so-called ranking algorithms [67], though PGMWM implements a sequential matching exploiting a deterministic order of non-RT containers, starting from PR to BE containers.

Illustrated in algorithm 1, the PGMWM consists of recurrently finding the maximal weight matching for each non-RT container $n \in \{L_p\}$, every timeslot, on a $\sum_{n\in L'_p} |c_n| \times M$ bipartite subgraph $G_n(V_n, E_n)$ between the set of CPUs required by non-RT container n denoted by $V_1 = c_n$ and the set of available CPUs $V_2 = \{1, 2, \ldots, M\}$, such that $V = V_1 \cup V_2$, and $V_1 \cap V_2 = \varnothing$. For each pair of vertices $n \in V_1^*$ and $m \in V_2$, there exists an edge $(n, m) \in E_n$ whose weight is given by $\omega_{nm}(t)$. A maximal matching is defined as the subset $E_n^* \in E_n$ containing the C_n edges (n, m) with the largest weights $\omega_{nm}(t)$. As shown in Appendix A, the the PGMWM finds the maximal matching for all non-RT containers in $O(\sum_{n\in L_p} |E_n^*| \log |E_n^*|)$.

Each iteration of the PGMWM algorithm results in a maximal weight matching that allocates CPUs to a non-RT container guaranteeing that its CPU requirements are met. With the new CPU allocation, PRINCIPIA updates the number of containers assigned to each CPU $\rho_m(\tau)$, each iteration in sub-timeslot granularity $\tau < t$. The counter $\rho_m(\tau)$ is used to compute the weight $\omega_{nm}(\tau)$ of each edge in the subgraph corresponding to the CPU allocation of the non-RT container in the next iteration of the algorithm, following 17. Computing $\omega_{nm}(\tau)$ using this iterative approach allows for the algorithm to provide evenly distributed CPU allocation among non-RT containers. Conversely, if computed at the beginning of timeslot t, $\rho_m(t)$ only reflects the containers mapped to CPUs until the previous slot, generating edge weights that favor allocation of CPUs with low congestion in the previous timeslot potentially leading to uneven CPU allocation.

Let $\chi_n^*(t) = (\chi_{n1}^*(t), \chi_{n2}^*(t), \ldots, \chi_{nM}^*(t))$ be the solution vector to CPU allocation problem in (10) following the PGMWM, where $\chi_{nm}^*(t)$ is given by:

$$\chi_{nm}^*(t) = \begin{cases} 1, & \text{if } (n, m) \in E_n^* \\ 0, & \text{otherwise} \end{cases} \tag{19}$$

On the other hand, rather than solving the CPU-shares subproblem as an LP, after allocating CPUs to all non-RT containers, PRINCIPIA computes the CPU-shares using a proportional allocation based on the CPU usage of each non-RT container. Let $S_n^*(t) = (S_{n1}^*(t), S_{n2}^*(t), \ldots, S_{nM}^*(t))$ be the CPU-shares decision vector of container n, where $S_{nm}^*(t)$ is computed as follows:

$$S_{nm}^*(t) = \frac{\alpha_n(t)}{\sum_{n\in L_p} \alpha_n(t)\chi_{nm}^*(t) + \sum_{r\in L_1} \alpha_r^*(t) I_{rm}} \tag{20}$$

PRINCIPIA allocates CPU-shares to containers sharing the same CPU proportionally as the ratio between the CPU intensity of container n over the sum of CPU intensities of containers mapped to the same CPU. While this proportional allocation benefits non-RT containers by avoiding CPU starvation, the CPU intensity prioritizes RT containers on the CPU-shares allocation.

Algorithm 1 *PRINCIPIA Greedy Maximal Weight Match*

Step_0 Input: $Q_n^*, \Omega_n, \alpha_n(t)$;
Initialization: $\overrightarrow{\chi_n} := 0$, $\overrightarrow{S_n} := 0$, $|\overrightarrow{\chi_n}| = \left|\overrightarrow{S_n}\right| = M$;
Step_1 **for** each $n \in L_p$
  for $k \in \{1, ..., C_n\}$
    $m := index(argmax\{\Omega_n\})$;
    $\overrightarrow{\chi_n}[m] := 1$;
    $\Omega_n[m] := 0$;
  end for
end for
Step_2 **for** $m \in \{1, 2, ..., M\}$
  $S_{nm} := \frac{\alpha_n(t)}{\sum_{n\in\{L_2\cup L_3\}} \alpha_n(t)\chi_{nm}^*(t) + \sum_{r\in L_1} \alpha_r^*(t) I_{rm}}$;
  $\overrightarrow{S_n}[m] = S_{nm}$;
end for

Step_3 $\chi_n^*(t) = \overrightarrow{\chi_n}$, $S_n^*(t) = \overrightarrow{S_n}$;
Step_4 Output: $\chi_n^*(t)$, $S_n^*(t)$;

D. Performance analysis

To assess the performance of PRINCIPIA for CPU and CPU-shares allocation, we simulate the PGMWM algorithm on an example CPU sharing scenario. This scenario comprises two RT containers, two PR containers, and two BE containers, as depicted in Figure 5. The PGMWM algorithm is evaluated against two optimal solutions: a GLPK based MIP solver (referred to as MIP) for the C2SAP in (5), and a custom solver designed specifically for the CPU and CPU-shares decomposition (referred to as MWM+LP) introduced above. Specifically, the decomposition solver comprises an instance of the Blosson algorithm [68] to solve the MWMP of the CPU allocation subproblem in (10), and a GPLK based LP solver to solve the CPU-shares allocation subproblem in (14). Additionally, a second CPU and CPU-shares decomposition solver (referred to as

MWM-SG+LP) is evaluated. Rather than solving the MWMP in an augmented bipartite graph, the MWM-SG+LP solves the MWMP recurrently solves the MWMP for each non-RT container on the same bipartite subgraphs implemented by PRINCIPIA, and a GPLK based LP solver to solve the resulting CPU-shares allocation subproblem. The idea behind this solver is to compare the performance of the PGMWM against the optimal solver in simple subgraphs.

Each container generates CPU-usage according to an independent and identically distributed (i.i.d) Normal process. Particularly, at the beginning of every slot, each container $n \in L$ generates i.i.d CPU-usage with probability $Q_n(t) \sim \mathcal{N}(\mu_n, \sigma_n^2)$, with parameters μ_n and σ_n^2. Similarly, at the beginning of every slot, each CPU m generates i.i.d CPU-availability with probability $G_m(t) \sim \mathcal{N}(\mu_n, \sigma_n^2)$, with parameters μ_m and $, \sigma_m^2$.

To compare the performance of PRINCIPIA and optimal methods for CPU and CPU-shares allocation, we evaluated solution metrics such as the achieved objective function value, approximation ratio, fairness index, and execution time. We computed the approximation ratio as the ratio between the mean objective function value computed by the approximation algorithm over slots and that of the optimal solution MIP. In essence, the approximation ratio provides a measure of how close the approximate solution obtained by a solver/algorithm is to the optimal solution. On the other hand, to assess how well these algorithms are balancing resource allocation across CPUs, we measure the Jain fairness index [69] of the amount of non-RT containers mapped to the CPUs. The Jain index provides a metric for evaluating how evenly non-RT containers are assigned to CPU resources, with values closer to 1 indicating a more even assignment of non-RT containers to the different CPUs. Furthermore, we also evaluate the fairness index of the CPU-shares allocation among the non-RT containers using the Jain index.

We conducted simulations over 100,000 timeslots. Table I shows the solution metrics obtained from the evaluation of PRINCIPIA and the optimal solvers. While the MIP and the MWM+LP provide optimal solutions, MWM-SG+LP and PRINCIPIA achieve approximations to the optimal solution with an average approximation ratio of 0.2. The reason for the relatively low approximation ratio is that PRINCIPIA and MWM-SG+LP recurrently compute the CPU allocation weights ω_{nm} as non-RT containers are allocated to the CPUs. Nevertheless, by updating the number of containers assigned to each CPU and recurrently computing the CPU allocation weights, PRINCIPIA and MWM-SG+LP perform a more even allocation of CPUs across non-RT containers, as reflected by their higher fairness indices compared to MIP and MWM+LP. Similarly, the fairness indices for CPU-shares allocation among non-RT containers are higher for PRINCIPIA compared to the optimal solvers, which reflects the proportional CPU-shares allocation approach adopted by PRINCIPIA.

The low fairness index values for the optimal solvers indicate that resources are not being evenly allocated. For instance, the allocation of CPUs alternates over slots in MIP and MWM+LP, as the number of containers assigned to each

Solution metric	Solver/Algorithm			
	MIP	MWM+LP	MWM-SG+LP	PGMWM
Objective function	7183.6	7184.2	1655.9	1624.6
Approximation ratio	1.0	1.0	0.23	0.22
Fairness index CPU assignment	0.5	0.5	0.78	0.78
Fairness index CPU-shares allocation	0.5	0.5	0.5	0.98

TABLE I: Solver/Algorithm performance. Evaluated solution metrics: (i) Objective function. (ii) Approximation ratio. (iii) Fairness index of allocation across CPUs. (iv) Fairness index of CPU-shares allocation.

CPU as well as the CPU allocation weights are updated at the beginning of each timeslot. Hence, the allocation prioritizes CPUs with high availability in the next slot. Similarly, for the CPU-shares allocation, the optimal solvers seek to maximize a weighted sum whose solution often falls in extreme points of the feasible region with the highest weights (i.e., RT containers receive higher CPU-shares weight as defined by the priority policy $k = 1$). As a result, the optimal solvers unevenly allocate CPU-shares among non-RT containers, leading to lower fairness indices compared to PRINCIPIA.

Finally, figure 4 provides an overview of the runtime for each solver. Certainly, PRINCIPIA's greedy approach to solving the CPU and CPU-shares subproblem is significantly faster than the optimal solvers, with a runtime in the microsecond range. In contrast, the MIP, MWM+LP and MWM-SG+LP solvers have significantly longer runtimes in the milisecond range, making them less suitable for practical MEC systems.

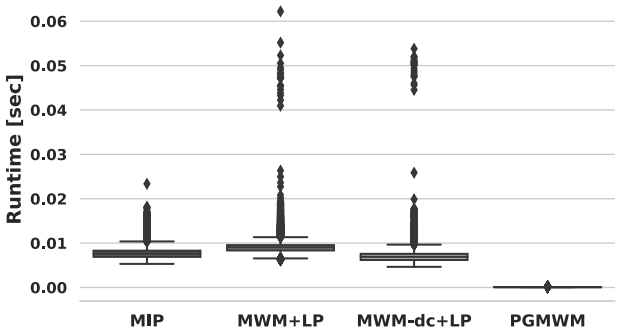


Fig. 4: Runtime profiles for all different solvers.

V. Assessing CPU sharing in MEC

This section evaluates CPU sharing in MEC servers that host RT and non-RT applications using containerized virtualization. Depicted in figure 5, the evaluation setup consists of a MEC server where two RT containers, two PR containers, and two BE containers share a set of four CPUs. The evaluation is composed of two parts. The first part, collocated RT containers, evaluates CPU sharing among collocated RT containers. The second part, collocated non-RT containers, evaluates CPU sharing when non-RT containers are collocated on the same CPUs as the RT containers.

Collocated RT containers: Assuming that both RT containers require two CPUs each, we evaluate two CPU sharing scenarios: (i) Sharing CPUs, where both RT containers run on the same two CPUs; and (ii) Orthogonal CPUs, where each

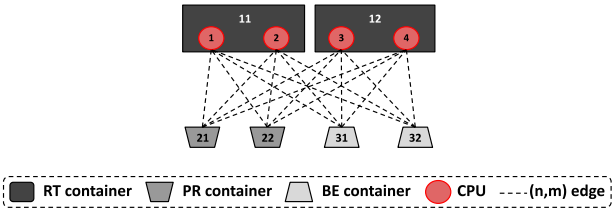


Fig. 5: CPU sharing. RT containers are allocated orthogonal CPUs. Both PR and BE run on CPUs allocated to RT container according to CPU sharing policies (e.g., PRINCIPIA).

RT container run on a set of two orthogonal CPUs. To emulate RT application workloads, each RT container runs a stressors of the synthetic benchmark tool *stress-ng*[4] with RT priority 99 (i.e., by setting the Linux RT attribute chrt = 99). The stressors consist of a single thread process instructed to rapidly change the CPU affinity. Switching this process's CPU affinity enables a scenario where the RT-Kernel schedules the process on the evaluated CPUs.

The conducted experiments consist of measuring the processing latency of RT containers in the Linux RT-Kernel. To do so, each RT container runs one thread of the cyclictest setting the RT priority 99. The cyclictest[5] provides an estimate of the system's real-time latency by measuring the difference between the time at which the thread signals to wake up and the wake up time. Here, each experiment captures the processing latency as reported by the cyclictest during a time span of 5 minutes.

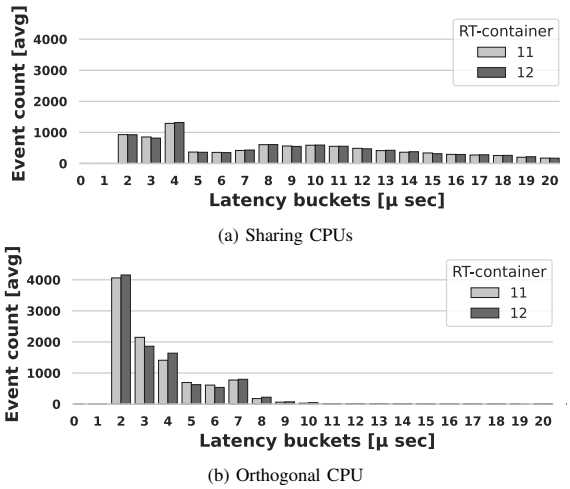
Figure 6 shows the distribution of processing latency events for both evaluated scenarios. These distributions compute the average of latency events over a set of twelve experiments for an observation time-span of 60 minutes. In the RT-Kernel, because a RT process can not preempt any other RT process, RT processes potentially spend longer time in a CPU runqueue waiting their time to run (i.e., processing interference) if several RT processes happen to be scheduled on the same CPU. Such is the case of the Sharing CPUs scenario where both RT containers run their processes on the same set of two CPUs. Consequently, RT containers often experience longer processing latency as shown by their tailed processing latency distributions. For instance, on average, while 75% of latency events fall on the first 15 μsec buckets, the maximum latency which is an indicator of the worst case execution time (WCET) spans until 0.1 to 0.3 msec.

To avoid processing interference from collocated RT containers, in the Orthogonal CPUs scenario, each RT container runs on a set of two different CPUs. The latency distribution for this scenario shows the benefits of running RT containers on orthogonal CPUs. Here, on average, while 75% of latency events fall in the first 4 μsec buckets, the maximum latency is bounded below 20 μsec. Based on this evidence, this paper adopts and encourages orthogonal CPU allocation for containers running time-sensitive applications in MEC.

Collocated non-RT containers: Aiming to avoid CPU underutilization from containers running on exclusive CPUs,

[4]https://wiki.ubuntu.com/Kernel/Reference/stress-ng

[5]https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start



(a) Sharing CPUs

(b) Orthogonal CPU

Fig. 6: Processing latency of RT containers. Computed over a set of twelve experiments (total observation time-span is 60 minutes) - Evaluated scenarios: (i) Sharing CPUs (RT container run on the same CPUs) (ii) Orthogonal CPUs (RT containers run on different CPUs).

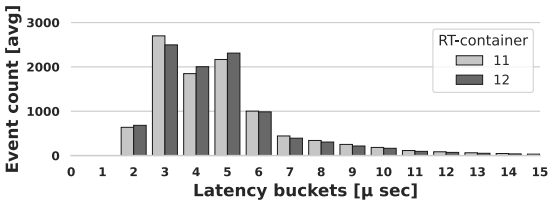
in this scenario, non-RT containers are collocated on the CPUs allocated to RT-containers. The methodology consists of evaluating the processing latency of deployed RT containers while sharing CPUs with collocated non-RT containers. As depicted in figure 5, let $L_1 = \{11, 12\}$ represent the set of RT containers whose CPU requirements are assumed as $C_{11} = C_{12} = 2$. Similarly, let $L_2 = \{21, 22\}$ represent the set of PR containers whose CPU requirements are assumed as $C_{21} = C_{22} = 2$. Finally, let $L_3 = \{31, 32\}$ represent the set of BE containers whose CPU requirements are assumed as $C_{31} = C_{32} = 2$. Assuming that RT containers are allocated orthogonal CPUs as indicated by $I_{11} = \{1, 1, 0, 0\}$ and $I_{12} = \{0, 0, 1, 1\}$, we evaluate two CPU sharing policies: (i) RT-Kernel, where the RT-Kernel schedules non-RT containers on the CPUs assigned to RT containers; (ii) PRINCIPIA, where the PRINCIPIA mechanism defines on which CPUs non-RT containers are scheduled by the RT-Kernel, and controls the amount of CPU-time that non-RT containers get granted on those CPUs.

To emulate various workloads, non-RT containers run different instances of the synthetic benchmark tool *stress-ng* stressing different physical resources. For instance, PR container 21 runs one stressors performing random memory read/write operations, and one virtual memory stressors writing up to 5GB to the allocated memory; PR container 22 runs two cache stressors that perform random widespread memory read and writes to thrash the CPU cache. Similarly, BE 31 container runs one virtual memory stressors writing up to 15GB to the allocated memory and one stressors continuously performing system calls map(2)/munmap(2)[6] (i.e., creates/deletes new mappings in the virtual address space) for up to 15GB; BE container 32 runs one stressors which performs asynchronous I/O writes using Linux system calls (e.g., io_setup, io_submit), one disk stressors which continually writes, reads and removes temporary files for up to 2GB, and one fork stressors which continually forks children processes that immediately exit.

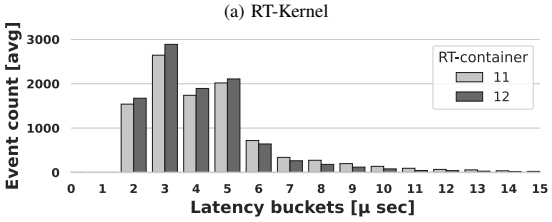
[6]https://manpages.ubuntu.com/manpages/bionic/man2/mmap.2.html

Finally, RT containers run the same workload as described in the previous part.

Consistent with the experimental approach used previously, the experiments measure the processing latency of RT containers in the Linux RT-Kernel while sharing computing resources with collocated non-RT containers. Figure 7 shows the distribution of processing latency measurements for both RT containers under the evaluated CPU sharing policies. These distributions are obtained by averaging latency events over a set of twelve experiments for an observation time-span of 60 minutes. By comparing these latency distributions with the results obtained for the Orthogonal CPUs scenario in the previous part (figure 9a), we can observe the impact of sharing computing resources with non-RT containers on the performance of RT containers. For example, latency events shift and spread across the 2 μsec bucket, generating tailed distributions. However, this impact is not as significant as it is for collocated RT containers, demonstrating the RT-Kernel's ability to prioritize RT processes. Nevertheless, an increase in processing latency occurs as a consequence of sharing MEC server's physical resources (e.g., memory, I/O).



(a) RT-Kernel



(b) PRINCIPIA

Fig. 7: Processing latency of RT containers. Computed over a set of twelve experiments (total observation time-span is 60 minutes) - Evaluated scenarios: (i) RT-Kernel (ii) PRINCIPIA.

Employing PRINCIPIA as a runtime resource management daemon in combination with the RT-Kernel can significantly reduce the impact of processing interference on RT containers, resulting in a decrease in processing latency. For example, as seen in Figure 7b, the event rate in the 2 μsec bucket is over 100% lower for the RT-Kernel scenario compared to PRINCIPIA. Additionally, PRINCIPIA effectively mitigates the impact on the worst-case execution time (WCET) of RT containers. This is demonstrated by the increase in the tail distribution of processing latency events presented in Figure 8, where PRINCIPIA achieves up to a 100% decrease in comparison with the RT-Kernel scenario.

To evaluate the potential for CPU-intensive RT containers to starve collocated non-RT containers, we measured the CPU usage of each container as a function of the target CPU usage, as shown in figure 9. Each container was subject to running as many CPU stressors of the *stress-ng* tool as its CPU requirement, while the RT containers executed their stressors

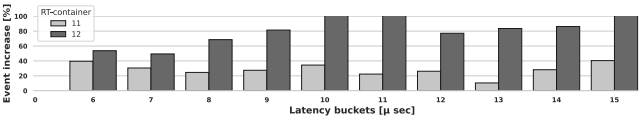
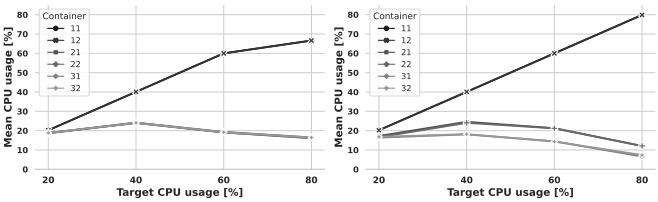


Fig. 8: Increase of tail processing latency events for the RT-Kernel scenario compared to PRINCIPIA.

with the RT priority chrt = 95 as they are intended to host RT applications.



(a) RT-Kernel (b) PRINCIPIA

Fig. 9: Mean CPU usage [%] as measured by each container $n \in L$.

CPU stressors running on RT containers are capable of preempting stressors on non-RT containers, resulting in RT containers achieving 100% processing throughput for most CPU targets. Conversely, in an overloaded system with an 80% target CPU usage, the RT-Kernel scenario had an 18% drop in the measured CPU compared to PRINCIPIA, which controls and prioritizes CPU-time allocation for RT containers.

Not only PRINCIPIA prioritizes RT containers, but also provides differentiated priority to PR and BE containers. As shown in figure 9b, PR containers perceive higher CPU usage than BE containers despite how overloaded the system is. Using the control constant υ_k, PRINCIPIA can control non-RT containers' influence on the CPU-shares allocation, which enables both differentiated allocation and a conservative mechanism to protect RT containers from potential processing interference from collocated workloads.

VI. Assessing CPU sharing in the Cloud-RAN

This section evaluates the Cloud-RAN while sharing computing resources in a MEC server deploying vBBUs along with collocated applications. To do so, this section studies the processing performance of vBBUs and the end-to-end traffic when the vBBUs share computing resources under different CPU sharing approaches.

A. Mobile Network scenario

To assess CPU sharing in the Cloud-RAN, we consider the mobile network scenario depicted in figure 10. Using Linux Containers (LXC), this network scenario deploys two vBBUs (referred to as vBBU1 and vBBU2). Each vBBU adopts a different functional split and is deployed between the DU and the CU. While each vBBU is the only process instantiated at the DU, both of these vBBUs share computing resources at the CU. Moreover, this network scenario considers a single UE connected to each of the vBBUs. The experimental setup deploying this mobile network is summarized in Appendix II.

Based on the LTE-BBU functional split model shown in figure 2, L1 refers to physical layer functions (e.g., both low

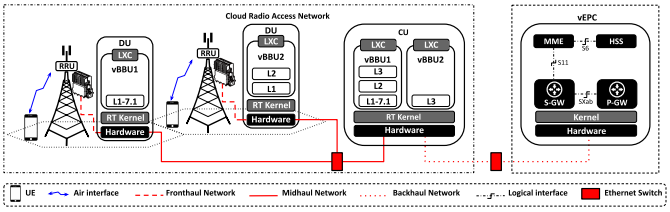


Fig. 10: Mobile network scenario based on the Cloud-RAN architecture. Deploying two vBBUs with split 7.1 and split 2, respectively. The mobile transport Xhaul is based on switched Ethernet.

PHY and high PHY), L2 refers to link layer functions (e.g., MAC and RLC), and L3 refers to the network layer functions (e.g., PDCP and RRC). When implemented in software, some of these functions are instantiated as RT processes. For instance, L1 and L2 functions. The reason is that these functions impose strict timing constraints, for example, to meet the HARQ deadline [5].

Adopting functional split 7.1, the vBBU1 deploys low PHY functions locally at the DU (i.e., a subset of L1 functions like the Fast Fourier Transform for UL, the Inverse Fast Fourier Transform for DL, and the add/remove cyclic prefix), while the CU deploys the remaining L1 functions, as well as L2 and L3 functions. Because vBBU1 deploys most of the RT functions at the CU, the MEC server hosting the CU instantiates the vBBU1 as an RT application on an RT container hence adopting the RT priority policy $k = 1$. On the other hand, vBBU2 adopts the functional split 2 deploying both L1 and L2 functions at DU, while the CU only instantiates L3 functions. As this functional split hosts all the functions involved in the HARQ loop (i.e., RT functions) at the DU, the CU instantiates the vBBU2 as non-RT prioritized application.

In this network scenario, the Fronthaul consists of a point-to-point link that transports digital samples of the RF signal. Deploying Ethernet as the mobile transport network for the MidHaul and Backhaul segments, the Midhau transports data where latency and caparacity requirements depend upon the chosen functional split. For instance, using split 7.1, the vBBU1 interchanges frequency domain of I/Q samples between the DU and the CU. Transmitting symbols in frequency domain reduces substantially the bandwidth requirement in the MidHaul network, in comparison with full centralization of vBBU functions at the CU (e.g., split 8) where time domain I/Q samples are transmitted between the DU and the CU. On the other hand, because vBBU2 hosts all time-sensitive functions at the DU, the data rate requirements in the MidHaul network is similar to that in the BackHaul. An Ethernet switch aggregates the mobile traffic from both vBBUs into an aggregation trunk link. Similarly, in the Backhaul, a second Ethernet switch aggregates the Backhaul traffic from each vBBU at the CU, connecting them to the vEPC.

B. CPU sharing at the CU

Figure 11 illustrates the CPU sharing scenario at the CU. Here, deployed containers share a set of $M = 4$ CPUs. Instantiated as an RT process, the vBBU1 runs on top of an RT container such that the set of RT containers is given by $L_1 = \{11\}$. Assuming that vBBU1 requires four CPUs for guaranteeing stable performance ($C_{11} = 4$), the RT container 11 is pre-allocated a set of four CPUs as indicated by $I_{11} = \{1, 1, 1, 1\}$.

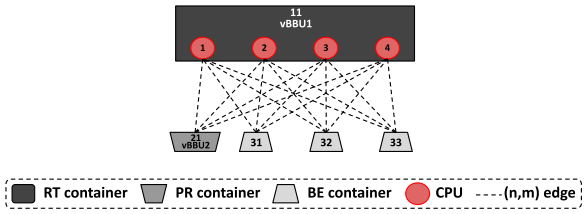


Fig. 11: CPU sharing at the CU. While vBBU1 is pre-allocated CPUs={1,2,3,4}, non-RT containers are collocated on use any of the CPUs allocated to vBBU1 subject to their CPU requirements.

The vBBU2 hosts L3 functions at the CU which perform control and data (e.g., GTP tunnel carrying UE's traffic) plane functions that are still critical functions for the mobile network and thus require prioritization on CPU allocation over other non-RT applications. As a result, the vBBU2 is instantiated on top of a PR container as given by the set of PR containers $L_2 = \{21\}$. Here, we assume that the CPU requirement of the PR container 22 is given by $C_{21} = 2$. On the other hand, non-RT applications run on top of BE containers as defined by the set of BE containers $L_3 = \{31, 32, 33\}$. Here, we assume that each BE container $n \in L_3$ has a CPU requirement given by $C_n = 2$.

C. Experimental design

Assuming that the RT container is allocated CPUs according to I_{11} in provisioning time, as shown in figure 11, the methodology consists of evaluating three CPU sharing scenarios: (i) Baseline, where both vBBU1 and vBUU2 run on isolated CPUs according to their CPU requirements. (ii) RT-Kernel, where the RT-Kernel schedules CPU-time to non-RT containers on the CPUs allocated to vBBU1. (iii) PRINCIPIA, where the PRINCIPIA algorithm allocates CPUs and CPU-shares to non-RT containers on the CPUs allocated to vBBU1. In this scenario, PRINCIPIA defines a time slot of 1 second, and a slicing window $W = 10$ for EMA computing. Furthermore, PRINCIPIA sets the control constants associated with the priority policy of each container, as follows: $\upsilon_1 = 1$, for RT containers; $\upsilon_2 = 2$, for PR containers; $\upsilon_3 = 3$, for BE containers.

Derived metrics: To evaluate the Cloud-RAN while sharing computing resources at the MEC server hosting the CU, this paper derives the following metrics:

1) **Scheduling latency:** is the amount of time a process spends waiting until obtaining CPU-time. This metric is the major contributor of a task processing latency [70]. In this paper, we measured the scheduling latency of the RT container instantiating vBBU1, using the Kernel tracing tool BPF Compiler Collection [71]. By measuring the Kernel run queue latency, this tool reports the time a task spends waiting its turn to run.
2) **Number of physical resource block retransmissions in Downlink (prbRetxDL):** as part of the DL Data Transmission Process (HARQ ACK/NACK) [72], the

UE performs error checking over received DL data. If an error has been found, the UE sends a DL-NACK to the vBBU. Then, the vBBU retransmits the DL data to the UE. The number of prbRetxDL, therefore, represents DL re-transmissions and is an important indicator of the health of the DL data path.

3) **End to end (e2e) TCP throughput:** this metric provides insight into the mobile network performance. Measuring the received throughput, each UE conducts a downstream TCP Iperf3 test to an Iperf3 server located at the SPGW-U IP interface at the vEPC.
4) **e2e latency:** derived from a round trip time (RTT) test, this metric provides insight on the mobile network latency. Measuring the RTT, each UE conducts a ICMP test to the SPGW-U IP interface at the EPC.

The conducted experiments aim to study the impact of CPU sharing on the Cloud-RAN performance. To do so, six experiments lasting ten minutes each were conducted for each CPU sharing scenario. During each experiment, the metrics described above were measured to gain insight into the performance of each scenario. To emulate UE's mobile traffic, two TCP downstream flows were generated during each experiment with a target rate of 13 Mbps per UE. Furthermore, to emulate non-RT application workload, each BE container ran an instance of the *stress-ng* synthetic benchmark tool which generated load on different MEC server resources. First, container 31 generates virtual memory stress consuming up to 5GB of memory. Second, container 32 consumes 200MB of cache. Finally, container 33 performs hard disk load for up to 2GB.

D. Results and Discussion

Figure 12 shows the histogram of vBBU1 scheduling latency events averaged over the set of experiments. The scheduling latency provides insights on the RT processing performance of RT processes running in the Linux RT-Kernel. More specifically, this figure shows the average of events that fall into different latency buckets. Here, we focus on analyzing vBBU1 as the RT application running on top of an RT container, which shares computing resources with collocated non-RT containers in a MEC server.

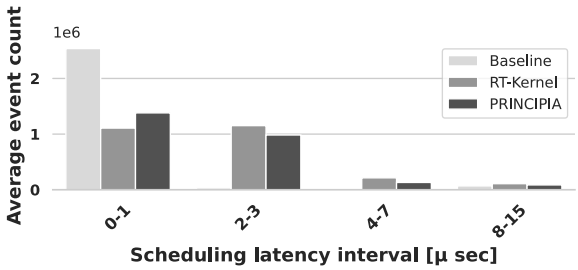


Fig. 12: vBBU1 histogram of scheduling latency events - Evaluated scenarios: (i) Baseline (ii) RT-Kernel (iii) PRINCIPIA.

In the Baseline scenario, the majority of the latency events fall into the 0-1 microsecond bucket, accounting for 96% of the latency events. Sharing computing resources with non-RT containers, however, introduces higher scheduling latency events across all the latency buckets. PRINCIPIA reduces the impact on the RT processing performance of vBBU1. For example, 53% of latency events occur within the 0-1 microsecond interval, and 42% fall within the 2-3 microsecond interval. Conversely, using the RT-Kernel, 44% of the scheduling latency events fall within the 0-1 microsecond interval (a 19% decrease compared to PRINCIPIA), and 44% fall within the 2-3 microsecond interval.

What is the impact of sharing computing resources with non-RT containers on the vBBU1's procedures? Figure 13 shows the average number of prbRetxDL, which is an indicator of the quality of the data path. In the RT-Kernel scenario, the number of prbRetxDL increases by 18% on average compared to the Baseline scenario. This increase shows that sharing computing resources with non-RT containers negatively impacts the vBBU1 procedures. In contrast, in PRINCIPIA, the prbRetxDL values increase by only 6% on average compared to the Baseline scenario, indicating that the PRINCIPIA CPU-sharing approach mitigates the negative impact on the data path quality.

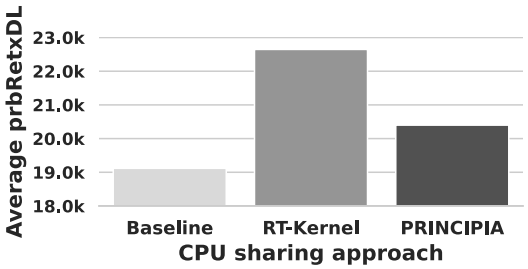
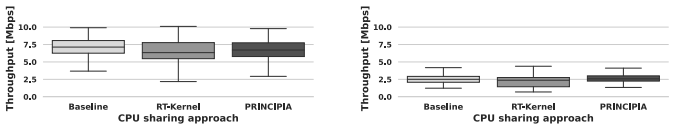


Fig. 13: vBBU1 number of physical resource block retransmissions in Downlink (prbRetxDL) - Evaluated scenarios: (i) Baseline (ii) RT-Kernel (iii) PRINCIPIA.

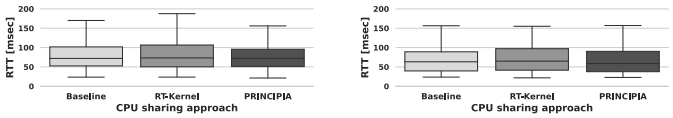
Because the user traffic server is located outside the EPC in the IP network, the e2e metrics measured by the UE in figure 10 provide insights into the overall mobile network performance rather than metrics for the MEC server's performance. Nevertheless, the e2e metrics reveal how sharing computing resources with vBBUs in MEC servers affects the mobile network's performance. Although the e2e capture metrics in the millisecond scale (while the RT performance analysis for resource sharing in MEC is in the microsecond), marginal analysis of the e2e metrics reveals how sharing computing resources with vBBUs in MEC servers affects the mobile network's performance. For example, figure 14 shows the average received throughput for the TCP downstream flow for each UE. Sharing computing resources between vBBUs in MEC servers generates variation in the received throughput, which can affect the stability and performance of the mobile network [73]. However, in the PRINCIPIA scenario, the variation in the received throughput is lower, and the distribution of variation is similar to that of the baseline scenario.

Similarly, the RTT results in figure 15 show that the PRINCIPIA scenario has a lower variability in RTT than the RT-Kernel, as evidenced by the interquartile range (IQR) of these RTT distributions. This variability reduction is similar to that observed in the baseline scenario, indicating that PRINCIPIA



(a) UE-vBBU1 (b) UE- vBBU2

Fig. 14: Cloud-RAN e2e performance - UE average TCP throughput. Target data rate is 13 Mbps



(a) UE-vBBU1 (b) UE- vBBU2

Fig. 15: Cloud-RAN e2e performance - UE mobile network latency through RTT

CPU sharing policy mitigates the impact of sharing computing resources on the mobile network latency.

VII. Related work

The vBBU on MEC: Running the vBBU on GPP managed by the Linux RT-Kernel has been widely studied in the literature. For instance, the Linux RTAI (Real Time Application Interface) [74], has been used in mobile system testbeds in [26], [75], [76]. On the other hand, the PREEMPT_RT patch has been used in few works in the context of Cloud-RAN [77], [78]. Included in the mainline code from the Ubuntu distribution, the Low-Latency Kernel patch [79] has been widely adopted by researchers using the OpenAirInterface (OAI) code [5], [80]–[82]. The main reason is that OAI's developers have optimized their code to provide full compatibility with the low-latency Kernel.

The works in [83] and [84] evaluated different virtualization environments and compared them with the bare-metal deployment: hypervisor, Docker containers, and Linux containers (LXC). Evidence from those works show that containers achieve lower processing time than hypervisor VMs. More specifically, LXC achieves similar processing time as the bare-metal deployment.

Sharing computing resource in MEC servers: Sharing physical resources (e.g., CPU, I/O, memory) among applications with diverse execution time requirements (e.g., mixed time-critically services) collocated on general purpose processors (GPP) have been widely investigated in the literature [32]. Only few works, though, have investigated the processing interference caused by Kernel space processing. For instance, Reghenzani et al. in [34] characterized the processing interference caused by different Kernel subsystems under different workloads of mixed time critical services.

Runtime resource management enables processing quality of service to time-critically applications while sharing computing resources with collocated workloads on GPP [40]. For instance, the work in [41] proposes adaptable runtime mapping of resources for RT applications running in embedded platforms along with collocated workloads. These ideas were extended in [42] to enable energy efficient vBBU processing, through a design model approach used in embedded systems which provides hybrid and flexible resource mapping. In MEC/Multi-Cloud computing, resource management considering time-critically services commonly relies on dynamic/static resource provisioning of RT VMs/containers [45]. For instance, the work in [85] proposes a dynamic resource provisioning mechanism for VMs running time-critical services, which prioritizes VMs according to their application deadlines. The work in [11] integrates these two paradigms to enable vBBU processing on a MEC server along with general purpose applications. An external mobile network controller computes and predicts worst case execution time (WCET) of vBBUs deployed on GPP, according to current performance and user traffic demands. Based on WCET predictions, the external controller dynamically defines the amount of CPUs allocated to the vBBU, while enabling collocated applications.

The resource management mechanism presented in this paper leverages the building block for containerized virtualization **Cgroups** through its capabilities to control and limit resources assigned to containers. A similar approach presented in [86] dynamically adjusts the number of CPUs allocated to containers according to each container's CPU-time demand and the system's load. To define the CPU limits for each container, the mechanism use **Cgroups** subsystems *cpu.quota* and *cpu.period* which are only visible by the non-RT Scheduler Completely Fair Scheduler (CFS). Similarly, the work in [87] proposes a mechanism which uses these two Cgroups subsystems aiming to allocating available CPU-time among collocated RT and non-RT containers. Although this approach is intended to prioritized CPU allocation to RT containers, using the non-RT Scheduler RT guarantees can not be provided. Conversely, in this paper, we consider a MEC system deploying the Linux RT-Kernel to provide RT guarantees to RT containers hosting both RT applications and the Cloud-RAN. In addition, we propose a CPU sharing mechanism based on leverage **Cgroups** subsystems **cpuset** and **CPU-shares** which allows non-RT containers to exploit underutilized CPUs allocated to RT containers.

VIII. Conclusions and future work

This paper proposes a CPU sharing mechanism for MEC servers hosting applications with RT and non-RT requirements using containerized virtualization. Based on heuristic solution of subproblems from the decomposition of a MIP, the proposed mechanism provides an efficient CPU sharing solution as demonstrated by its outperformance of optimal solvers in terms of runtime. Furthermore, the recurrent CPU allocation and proportional CPU-shares approach allow for a fair resource allocation.

Through an empirical approach, this paper investigated the impact on the vBBU processing performance caused by sharing computing resources with collocated applications in MEC server. To mitigate the impact of collocated workloads and improve the vBBU performance, this paper proposed and evaluates a CPU sharing mechanism that runs as a user-space process (e.g., daemon) and operates in conjunction with the Linux RT-Kernel. First, this mechanism assumes that RT applications including the vBBUs are assigned orthogonal CPUs, as

a strategy to avoid processing interference with collocated RT (non-preemptable) processes. Then, by monitoring processor CPU utilization and container CPU usage as system metrics of resource availability and user demands, this mechanism enables non-RT containers on the same CPU allocated to RT containers. By controlling the amount of CPU-time that those containers are allowed to use on Cpus allocated to RT containers, this mechanism aims to mitigate the impact that sharing computing resources causes on RT applications.

Conducted evaluation on a MEC server deploying a combination of RT and non-RT containers shows that our CPU sharing mechanism outperforms the default RT-Kernel in mitigating the impact of resources sharing on RT applications. Using our CPU sharing mechanism the WCET is reduced by more than 150% in comparison with the default RT-Kernel approach. This evidence is strengthened when assessing the use of this CPU-sharing mechanism on the Cloud-RAN, where vBBUs share resources with collocated applications in MEC server. Using this mechanism, scheduling latency events of running the vBBU in the RT-Kernel decreases by up to 21% in comparison with the RT-Kernel approach.

This study could be extended to include sharing computation among different cells, for example, joint transmission and reception, coordinated scheduling, among others. Also, to include scenarios that consider migrating RAN functions across MEC servers, for example, using dynamic flexible functional splittings [88].

REFERENCES

[1] N. Bhushan, J. Li, D. Malladi, R. Gilmore, D. Brenner, A. Damnjanovic, R. T. Sukhavasi, C. Patel, and S. Geirhofer, "Network densification: the dominant theme for wireless evolution into 5g," *IEEE Communications Magazine*, vol. 52, no. 2, pp. 82–89, 2014.

[2] X. Ge, S. Tu, G. Mao, C.-X. Wang, and T. Han, "5g ultra-dense cellular networks," *IEEE Wireless Communications*, vol. 23, no. 1, pp. 72–79, 2016.

[3] A. Checko, H. L. Christiansen, Y. Yan, L. Scolari, G. Kardaras, M. S. Berger, and L. Dittmann, "Cloud ran for mobile networks—a technology overview," *IEEE Communications Surveys Tutorials*, vol. 17, no. 1, pp. 405–426, 2015.

[4] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing—a key technology towards 5g," *ETSI white paper*, vol. 11, no. 11, pp. 1–16, 2015.

[5] N. Nikaein, "Processing radio access network functions in the cloud: Critical issues and modeling," in *Proceedings of the 6th International Workshop on Mobile Cloud Computing and Services*, ser. MCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 36–43. [Online]. Available: https://doi.org/10.1145/2802130.2802136

[6] V. Struhár, M. Behnam, M. Ashjaei, and A. V. Papadopoulos, "Real-Time Containers: A Survey," in *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)*, ser. OpenAccess Series in Informatics (OASIcs), A. Cervin and Y. Yang, Eds., vol. 80. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, pp. 7:1–7:9. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/12001

[7] L. Abeni, A. Balsini, and T. Cucinotta, "Container-based real-time scheduling in the linux kernel," *SIGBED Rev.*, vol. 16, no. 3, p. 33–38, nov 2019. [Online]. Available: https://doi.org/10.1145/3373400.3373405

[8] L. Liu, H. Wang, A. Wang, M. Xiao, Y. Cheng, and S. Chen, "Mind the gap: Broken promises of cpu reservations in containerized multi-tenant clouds," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 243–257. [Online]. Available: https://doi.org/10.1145/3472883.3486997

[9] R. Delgado and B. W. Choi, "New insights into the real-time performance of a multicore processor," *IEEE Access*, vol. 8, pp. 186 199–186 211, 2020.

[10] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang *et al.*, "{PerfIso}: Performance isolation for commercial {Latency-Sensitive} services," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 519–532.

[11] X. Foukas and B. Radunovic, "Concordia: Teaching the 5g vran to share compute," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 580–596. [Online]. Available: https://doi.org/10.1145/3452296.3472894

[12] A. Checko, H. L. Christiansen, Y. Yan, L. Scolari, G. Kardaras, M. S. Berger, and L. Dittmann, "Cloud ran for mobile networks—a technology overview," *IEEE Communications Surveys Tutorials*, vol. 17, no. 1, pp. 405–426, 2015.

[13] G. T. 38.801, "Study on new radio access technology: Radio access architecture and interfaces," 2017.

[14] IEEE, "Ieee std 1914.1-2019: Standard for packet-based fronthaul transport network," *IEEE Standards*, 2019. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9079731

[15] L. M. P. Larsen, A. Checko, and H. L. Christiansen, "A survey of the functional splits proposed for 5g mobile crosshaul networks," *IEEE Communications Surveys Tutorials*, vol. 21, no. 1, pp. 146–172, 2019.

[16] F. W. Murti, J. A. Ayala-Romero, A. Garcia-Saavedra, X. Costa-Pérez, and G. Iosifidis, "An optimal deployment framework for multi-cloud virtualized radio access networks," *IEEE Transactions on Wireless Communications*, vol. 20, no. 4, pp. 2251–2265, 2021.

[17] P. Assimakopulos, G. S. Birring, M. K. Al-Hares, and N. J. Gomes, "Ethernet-based fronthauling for cloud-radio access networks," in *2017 19th International Conference on Transparent Optical Networks (ICTON)*, 2017, pp. 1–4.

[18] N. J. Gomes, P. Sehier, H. Thomas, P. Chanclou, B. Li, D. Munch, P. Assimakopoulos, S. Dixit, and V. Jungnickel, "Boosting 5g through ethernet: How evolved fronthaul can take next-generation mobile to the next level," *IEEE Vehicular Technology Magazine*, vol. 13, no. 1, pp. 74–84, 2018.

[19] L. Tomaszewski, S. Kukliński, and R. Kołakowski, "A new approach to 5g and mec integration," in *Artificial Intelligence Applications and Innovations. AIAI 2020 IFIP WG 12.5 International Workshops*, I. Maglogiannis, L. Iliadis, and E. Pimenidis, Eds. Cham: Springer International Publishing, 2020, pp. 15–24.

[20] A. Mosnier, "Embedded/real-time linux survey," 2005. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.133.6318&rep=rep1&type=pdf

[21] M. Timmerman, "Real-time capabilities in the standard linux kernel: How to enable and use them?" *International Journal on Recent and Innovation Trends in Computing and Communication*, vol. 3, no. 1, pp. 131–135, 2015.

[22] V. Yodaiken *et al.*, "The rtlinux manifesto," in *Proc. of the 5th Linux Expo*, 1999.

[23] I. Molnar, "Linux low latency patch," Last accessed Dec, 2021. [Online]. Available: https://web.archive.org/web/20080306131124/http://www.zipworld.com.au/~akpm/linux/schedlat.html

[24] T. L. Foundation, "Preempt_rt patch," https://wiki.linuxfoundation.org/realtime/preempt_rt_versions.

[25] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time linux kernel: A survey on preempt_rt," *ACM Comput. Surv.*, vol. 52, no. 1, p. 36, feb 2019.

[26] N. Nikaein, R. Knopp, F. Kaltenberger, L. Gauthier, C. Bonnet, D. Nussbaum, and R. Ghaddab, "Openairinterface: an open lte network in a pc," in *Proceedings of the 20th annual international conference on Mobile computing and networking*, 2014, pp. 305–308.

[27] H. Kim and R. R. Rajkumar, "Predictable shared cache management for multi-core real-time virtualization," vol. 17, no. 1, 2017. [Online]. Available: https://doi.org/10.1145/3092946

[28] S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. Gill, O. Sokolsky, and I. Lee, "Real-time multi-core virtual machine scheduling in xen," in *2014 International Conference on Embedded Software (EMSOFT)*, 2014, pp. 1–10.

[29] C. Pahl, "Containerization and the paas cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.

[30] Z. Li, M. Kihl, Q. Lu, and J. A. Andersson, "Performance overhead comparison between hypervisor and container based virtualization," in *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, 2017, pp. 955–962.

[31] R. Cavicchioli, N. Capodieci, and M. Bertogna, "Memory interference characterization between cpu cores and integrated gpus in mixed-criticality platforms," in *2017 22nd IEEE International Conference on*

Emerging Technologies and Factory Automation (ETFA), 2017, pp. 1–10.

[32] A. Burns and R. I. Davis, "Mixed criticality systems-a review:(february 2022)," 2022.

[33] P. De, V. Mann, and U. Mittaly, "Handling os jitter on multicore multithreaded systems," in *2009 IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1–12.

[34] F. Reghenzani, G. Massari, and W. Fornaciari, "Mixed time-criticality process interferences characterization on a multicore linux system," in *2017 Euromicro Conference on Digital System Design (DSD)*, 2017, pp. 427–434.

[35] M. Barletta, M. Cinque, L. De Simone, and R. Della Corte, "Achieving isolation in mixed-criticality industrial edge systems with real-time containers," in *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[36] S. Shekhar and A. Gokhale, "Dynamic resource management across cloud-edge resources for performance-sensitive applications," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017, pp. 707–710.

[37] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: Survey of current and emerging trends," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–10.

[38] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating interference at microsecond timescales," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'20. USA: USENIX Association, 2020.

[39] W. Fornaciari, G. Pozzi, F. Reghenzani, A. Marchese, and M. Belluschi, "Runtime resource management for embedded and hpc systems," ser. PARMA-DITAM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 31–36. [Online]. Available: https://doi.org/10.1145/2872421.2893173

[40] M. Niknafs, I. Ukhov, P. Eles, and Z. Peng, "Runtime resource management with workload prediction," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3316781.3317902

[41] R. Khasanov and J. Castrillon, "Energy-efficient runtime resource management for adaptable multi-application mapping," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 909–914.

[42] R. Khasanov, J. Robledo, C. Menard, A. Goens, and J. Castrillon, "Domain-specific hybrid mapping for energy-efficient baseband processing in wireless networks," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5s, sep 2021. [Online]. Available: https://doi.org/10.1145/3476991

[43] S. S. Manvi and G. Krishna Shyam, "Resource management for infrastructure as a service (iaas) in cloud computing: A survey," *Journal of Network and Computer Applications*, vol. 41, pp. 424–440, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804513002099

[44] M. P. Alves, F. C. Delicato, I. L. Santos, and P. F. Pires, "Lw-coedge: a lightweight virtualization model and collaboration process for edge computing," *World Wide Web*, vol. 23, no. 2, pp. 1127–1175, 2020.

[45] M. Azarmipour, H. Elfaham, J. Grothoff, C. von Trotha, C. Gries, and U. Epple, "Dynamic resource management for virtualization in industrial automation," in *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*, 2018, pp. 2878–2883.

[46] T. V. Doan, G. T. Nguyen, H. Salah, S. Pandi, M. Jarschel, R. Pries, and F. H. P. Fitzek, "Containers vs virtual machines: Choosing the right virtualization technology for mobile edge cloud," in *2019 IEEE 2nd 5G World Forum (5GWF)*, 2019, pp. 46–52.

[47] S. Hansun, "A new approach of moving average method in time series analysis," in *2013 Conference on New Media Studies (CoNMedia)*, 2013, pp. 1–4.

[48] C. C. Holt, "Forecasting seasonals and trends by exponentially weighted moving averages," *International Journal of Forecasting*, vol. 20, no. 1, pp. 5–10, 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169207003001134

[49] C. Brown, *Technical Analysis for the trading Professional*. McGraw Hill Professional, 1999.

[50] P. J. Kaufman, *Trading Systems and Methods,+ Website*. John Wiley & Sons, 2013, vol. 591.

[51] G. F. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems: concepts and design*. pearson education, 2005.

[52] N. Altay, P. E. Robinson, and K. M. Bretthauer, "Exact and heuristic solution approaches for the mixed integer setup knapsack problem," *European Journal of Operational Research*, vol. 190, no. 3, pp. 598–609, 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377221707006492

[53] A. Lodi, *Mixed Integer Programming Computation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 619–645. [Online]. Available: https://doi.org/10.1007/978-3-540-68279-0_16

[54] R. Rahmaniani, T. G. Crainic, M. Gendreau, and W. Rei, "The benders decomposition algorithm: A literature review," *European Journal of Operational Research*, vol. 259, no. 3, pp. 801–817, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377221716310244

[55] A. Gerards, "Chapter 3 matching," in *Network Models*, ser. Handbooks in Operations Research and Management Science. Elsevier, 1995, vol. 7, pp. 135–224. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0927050705801203

[56] H. N. Gabow and R. E. Tarjan, "Faster scaling algorithms for network problems," *SIAM Journal on Computing*, vol. 18, no. 5, pp. 1013–1036, 1989.

[57] Z. Galil, "Efficient algorithms for finding maximum matching in graphs," *ACM Comput. Surv.*, vol. 18, no. 1, p. 23–38, mar 1986. [Online]. Available: https://doi.org/10.1145/6462.6502

[58] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800020109

[59] J. Edmonds, "Maximum matching and a polyhedron with 0, 1-vertices," *Journal of research of the National Bureau of Standards B*, vol. 69, no. 125-130, pp. 55–56, 1965.

[60] R. Duan and S. Pettie, "Linear-time approximation for maximum weight matching," *J. ACM*, vol. 61, no. 1, jan 2014. [Online]. Available: https://doi.org/10.1145/2529989

[61] R. Duan and H.-H. Su, *A Scaling Algorithm for Maximum Weight Matching in Bipartite Graphs*, pp. 1413–1424. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611973099.111

[62] Z. G. Tang, X. Wu, and Y. Zhang, "Towards a better understanding of randomized greedy matching," in *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, ser. STOC 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1097–1110. [Online]. Available: https://doi.org/10.1145/3357713.3384265

[63] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[64] C. Joo, X. Lin, and N. B. Shroff, "Greedy maximal matching: Performance limits for arbitrary network graphs under the node-exclusive interference model," *IEEE Transactions on Automatic Control*, vol. 54, no. 12, pp. 2734–2744, 2009.

[65] N. Arnosti, "Greedy matching in bipartite random graphs," *Stochastic Systems*, vol. 12, no. 2, pp. 133–150, 2022. [Online]. Available: https://doi.org/10.1287/stsy.2021.0082

[66] R. Agarwal, S. Rajakrishnan, and D. B. Shmoys, "From switch scheduling to datacenter scheduling: Matching-coordinated greed is good," in *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, ser. PODC'22. New York, NY, USA: Association for Computing Machinery, 2022, p. 313–323. [Online]. Available: https://doi.org/10.1145/3519270.3538443

[67] R. M. Karp, U. V. Vazirani, and V. V. Vazirani, "An optimal algorithm for on-line bipartite matching," in *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, ser. STOC '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 352–358. [Online]. Available: https://doi.org/10.1145/100216.100262

[68] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.

[69] R. K. Jain, D.-M. W. Chiu, W. R. Hawe *et al.*, "A quantitative measure of fairness and discrimination," *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, vol. 21, 1984.

[70] F. Cerqueira and B. Brandenburg, "A comparison of scheduling latency in linux, preempt-rt, and litmus rt," in *9th Annual workshop on operating systems platforms for embedded real-time applications*. SYSGO AG, 2013, pp. 19–29.

[71] IO-Visor, "Bpf compiler collection (bcc)," Last accessed November, 2021. [Online]. Available: https://github.com/iovisor/bcc

[72] 3GPP, "Evolved universal terrestrial radio access (e-utra); medium access control (mac) protocol specification," 2007. [Online]. Available: https://www.arib.or.jp/english/html/overview/doc/STD-T104v4_10/5_Appendix/Rel13/36/36321-d20.pdf

[73] M.-R. Fida, M. Roald, E. Acar, and A. Elmokashfi, "Modeling variation in mobile download speed in presence of missing samples," *IEEE Transactions on Mobile Computing*, pp. 1–16, 2022.

[74] G. Giacobbi, "The gnu netcat project," Last accessed Nov, 2021. [Online]. Available: http://netcat.sourceforge.net

[75] F. Kaltenberger and S. Wagner, "Experimental analysis of network-aided interference-aware receiver for lte mu-mimo," in *2014 IEEE 8th Sensor Array and Multichannel Signal Processing Workshop (SAM)*, June 2014, pp. 325–328.

[76] I. Alyafawi, E. Schiller, T. Braun, D. Dimitrova, A. Gomes, and N. Nikaein, "Critical issues of centralized and cloudified lte-fdd radio access networks," in *2015 IEEE International Conference on Communications (ICC)*. IEEE, 2015, pp. 5523–5528.

[77] S. Bhaumik, S. P. Chandrabose, M. K. Jataprolu, G. Kumar, A. Muralidhar, P. Polakos, V. Srinivasan, and T. Woo, "Cloudiq: A framework for processing base stations in a data center," in *Proceedings of the 18th annual international conference on Mobile computing and networking*. ACM, 2012, pp. 125–136.

[78] I. Fajjari, N. Aitsaadi, and S. Amanou, "Optimized resource allocation and rrh attachment in experimental sdn based cloud-ran," in *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, 2019, pp. 1–6.

[79] I. Molnar, "Linux low latency patch," Last accessed Dec, 2021. [Online]. Available: https://web.archive.org/web/20080306131124/http://www.zipworld.com.au/~akpm/linux/schedlat.html

[80] S.-C. Huang, Y.-C. Luo, B.-L. Chen, Y.-C. Chung, and J. Chou, "Application-aware traffic redirection: A mobile edge computing implementation toward future 5g networks," in *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)*, 2017, pp. 17–23.

[81] A. Younis, T. X. Tran, and D. Pompili, "Bandwidth and energy-aware resource allocation for cloud radio access networks," *IEEE Transactions on Wireless Communications*, vol. 17, no. 10, pp. 6487–6500, 2018.

[82] X. Foukas, N. Nikaein, M. M. Kassem, M. K. Marina, and K. Kontovasilis, "Flexran: A flexible and programmable platform for software-defined radio access networks," in *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 427–441. [Online]. Available: https://doi.org/10.1145/2999572.2999599

[83] N. Nikaein, E. Schiller, R. Favraud, R. Knopp, I. Alyafawi, and T. Braun, *Towards a Cloud-Native Radio Access Network*. Cham: Springer International Publishing, 2017, pp. 171–202. [Online]. Available: https://doi.org/10.1007/978-3-319-45145-9_8

[84] C.-N. Mao, M.-H. Huang, S. Padhy, S.-T. Wang, W.-C. Chung, Y.-C. Chung, and C.-H. Hsu, "Minimizing latency of real-time container cloud for software radio access networks," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015, pp. 611–616.

[85] R. Begam, W. Wang, and D. Zhu, "Timer-cloud: Time-sensitive vm provisioning in resource-constrained clouds," *IEEE Transactions on Cloud Computing*, vol. 8, no. 1, pp. 297–311, 2020.

[86] H. Huang, Y. Zhao, J. Rao, S. Wu, H. Jin, D. Wang, K. Suo, and L. Pan, "Adapt burstable containers to variable cpu resources," *IEEE Transactions on Computers*, pp. 1–1, 2022.

[87] J. Wu and T.-I. Yang, "Dynamic cpu allocation for docker containerized mixed-criticality real-time systems," in *2018 IEEE International Conference on Applied System Invention (ICASI)*, 2018, pp. 279–282.

[88] T. Pamuklu, M. Erol-Kantarci, and C. Ersoy, "Reinforcement learning based dynamic function splitting in disaggregated green open rans," in *ICC 2021 - IEEE International Conference on Communications*, 2021, pp. 1–6.

[89] A. Rădulescu and A. J. Van Gemund, "On the complexity of list scheduling algorithms for distributed-memory systems," in *Proceedings of the 13th international conference on Supercomputing*, 1999, pp. 68–75.

[90] OAI, "F1 interface," Last accessed Dec, 2021. [Online]. Available: https://gitlab.eurecom.fr/oai/openairinterface5g/wikis/f1-interface

[91] 3GPP, "5g; ng-ran; f1 application protocol (f1ap) (3gpp ts 38.473 version 15.2.1 release 15)," Last accessed Nov, 2021. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/138400_138499/138473/15.02.01_60/ts_138473v150201p.pdf

[92] J. Claassen, R. Koning, and P. Grosso, "Linux containers networking: Performance and scalability of kernel modules," in *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2016, pp. 713–717.

APPENDIX A
PRINCIPIA TIME COMPLEXITY

The time complexity of PRINCIPIA would depend on the number of iterations required for the algorithm (1) to converge to a satisfactory solution. A step-by-step breakdown of the PRINCIPIA algorithm and its time complexity analysis is as follows:

1) In step one, the algorithm solves the CPU allocation problem. Here, the PGMWM selects C_n links in the bipartite graph $G_n(V_n*, E_n^*)$ in decreasing order. The time complexity of the PGMWM algorithm primarily depends on sorting and finding the C_n edges in the bipartite graph in decreasing order based on their weights. Typically, the time complexity of selecting in decreasing order is $O(|E_n^*| \log |E_n^*|)$ [89]. Because the PGMWM solves the CPU allocation problem recurrently for $n \in L_p$, the aggregated time complexity is $O(\sum_{n \in L_p} |E_n^*| \log |E_n^*|)$.
2) In step two, the algorithm allocates CPU-shares to each container on allocated CPUs based on the CPU allocations from step one. PRINCIPIA allocates CPU-shares proportionally to the CPU usage of containers allocated to the same CPU. Thus, PRINCIPIA iterates over M CPUs. In the worst case, PRINCIPIA computes the CPU shares for each container $n \in L$ in each CPU $m = \{1, 2, ..., M\}$. Thus, the time complexity is given by $O(|L| \times M)$.

Therefore, the overall time complexity of the PRINCIPIA algorithm for solving the CPU and CPU-shares allocation problem is $O(\sum_{n \in L_p} |E_n^*| \log |E_n^*| + |L| \times M)$.

APPENDIX B
TESTBED SETUP SPECIFICATION

Table II summarizes the software and hardware specifications, which are used to deploy the experimental setup following the mobile network scenario depicted in figure 10.

In containerized virtualization, the NIC is likely shared among several containers. In this network scenario, both vBBUs share a two port NIC supporting SR-IOV (see the hardware details in table II). SR-IOV enables sharing the resources of a NIC - PCI Express (PCIe) device. To do so, the NIC's PCI function (PF) is partitioned into several virtual functions (VF). When defining a VF, the NIC's driver supporting SR-IOV registers the corresponding RX/TX Hard-IRQ for that VF. Moreover, a unique MAC address is assigned to each VF. This pair of MAC address - RX/TX Hard IRQ makes the VF look like an independent NIC itself. Assigned to one of such VF, an LXC can access the network with complete traffic isolation. Previous evidence suggests that mechanisms based on creating virtual NIC (vNIC) like macvlan or SR-IOV provide lower overhead than Kernel based software switch mechanisms like Linux bridge or OVS [92]. Here, while vBBU use one of the NIC's port to access the Midhaul, the second port is used to access the Backhaul. By using two different port for Midhaul and Backhaul, the SR-IOV NIC enables isolating the Midhaul traffic with RT constrains from the best-effort Backhaul traffic.

Component	Description
UE	OnePlus-5 phone
air interface (as specified by the vBBUs)	25 Physical Resource Blocks (PRB), which provides 5 MHz bandwidth.
RRUs	Ettus (B210) Universal Software Radio Peripheral (USRP) platform. One antenna port - Single Input Single Output (SISO).
Fronthaul	Fast SuperSpeed USB 3.0 connectivity at 5.0 Gbit/s (provided by Ettus (B210)).
DU	Intel NUC7i7BNB equipped with four Intel Core i7-7567U processors, and 32 GiB of memory
DU's OS	Ubuntu 16.04 with low-latency Linux kernel version 4.19.58.
Midhaul & Backhaul	Juniper EX4200 Ethernet switch, with physical interfaces at 1 Gbit/s.
vBBU1	OpenAirInterface eNodeB-LTE implementation with split 7.1. This functional split uses the NGFI-IF4p5 interface specification [79].
vBBU2	penAirInterface eNodeB-LTE implementation with split 2 using the F1 Application Protocol (F1AP) [90], [91].
CU	GPP equipped with eight Intel i7-8750H physical processors at 2.20 GHz, and 32 GiB of memory.
CU's NIC	Supermicro AOC-SG-i2 Gigabit Ethernet adapter, equipped with two Intel 82575 Gigabit Ethernet ports.
vEPC	4G EPC implementation from OpenAirInterface
vEPC's host physical machine	GPP equipped with four Intel i7 processor at 2.20GHz, and 12 GiB of memory.
vEPC's host OS	Ubuntu 18.04 with generic Linux kernel version 5.3.28.

TABLE II: Cloud-RAN testebed setup - hardware and software specifications.