PRINCIPIA: Opportunistic CPU and CPU-shares Allocation for Containerized Virtualization in Mobile Edge Computing

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, Mobile Edge Computing (MEC) deploys multiple services with different execution time requirements running as isolated processes. For instance, both real-time (RT) and non-RT applications may be (are) running on the same infrastructure using containerized virtualization. Nevertheless, sharing resources (e.g., CPU) with collocated workloads could impact the RT performance of RT applications. This paper presents PRINCIPIA, a dynamic CPU and CPU-shares allocation mechanism that opportunistically enables non-RT applications to run on underutilized CPUs while providing RT guarantees to RT applications. By monitoring MEC's system metrics like processor's CPU utilization and container's CPU usage, PRINCIPIA dynamically allocates both CPU and CPU-shares to containers running non-RT applications aiming at opportunistically exploiting underutilized CPUs by containers running RT applications. We evaluate PRINCIPIA on a small-scale MEC server which uses containerized virtualization along with Linux RT Kernel to deploy both RT and non-RT applications. Our findings show that PRINCIPIA mitigates the impact on the RT performance of RT applications providing bounded processing latency in comparison with the default host Kernel scheduler.**

***Index Terms*—Mobile Edge Computing, Virtualization, Containers, Real-time containers, CPU sharing, CPU allocation**

I. Introduction

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 [1]. For instance, MEC caters for a wide spectrum of services with diverse requirements often imposing low-latency constraints. According to their execution time requirements, MEC servers run mobile services either as RT or non-RT processes. To meet the execution time requirements of RT services, both the MEC's operating system (OS) and the virtualization environment must provide RT guarantees.

While running RT processes on a MEC system, RT processes likely share computing resources (e.g., CPU time, I/O, memory) either with collocated user-space processes or with Kernel threads. Depending on the workload, sharing resources can impact the performance of RT applications [2]. To avoid processing interference from collocated processes, a common approach in RT systems (commonly deployed on embedded systems) is to run RT processes on a set of isolated CPUs. Nevertheless, isolating CPUs increases CPU underutilization [3]. Because the MEC hosts multiple applications with different execution time requirements, there is a need for efficient CPU sharing mechanisms among collocated applications using virtualization technology while providing real-time guarantees [4].

Understanding the interplay between the benefits of sharing computing resources and the impact on the RT performance of mobile services, allows to provide guidelines for achieving robust MEC. This paper presents PRINCIPIA, a CPU sharing mechanism for containerized virtualization in MEC. PRINCIPIA enables non-RT services to exploit the underutilized CPU-time allocated to RT services. To avoid processing interference from collocated workloads, PRINCIPIA proposes inverse constant square factors to prioritize containers running RT applications. Besides allocating CPUs, PRINCIPIA also allocates CPU-shares to containers. CPU-shares defines the relative amount of CPU-time that a given container is allowed to use.

In summary, the main contributions of this paper include:

- developing CPU sharing policies for containerized virtualization in MEC,
- using the notion of inverse constant square factors to control CPU and CPU-shares allocation, and
- evaluating the realization of RT services while adopting PRINCIPIA for CPU resource sharing.

The rest of the paper is organized as follows: section II presents the background and related work on containerized virtualization to run both RT and non-RT applications in MEC. Section III models the MEC system, 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. Section V describes the conducted experiments and results, while section VI concludes the paper.

II. Background and Related work

This section covers the background and discusses the use of containerized virtualization along with Linux RT Kernel to deploy both RT and not-RT applications on the same MEC system. Referred to as operating-system-level virtualization, containerized virtualization enables running several applications as isolated processes on the same system by using Kernel's features **Cgroups** and **namespaces** [5]. While Cgroups enables defining the limits on the use of system resources (e.g., CPU-time, memory, network bandwidth) to user-defined groups of processes or tasks, namespaces enables isolating the resources seen by a group of processes.

A. Running RT applications on Containerized virtualization

In computing, a real-time (RT) application is defined by the upper-bound execution time constraint (i.e., deadline) in which the application should run [6]. The way the system behaves when a deadline is missed, defines the classification of the RT application (e.g., soft-RT, hard-RT).

Linux RT Kernel as MEC's OS running RT applications: To run RT applications on top of a MEC server, the host OS must provide RT guarantees: preemption and a scheduling policy that focuses on meeting timing constraints of individual processes rather than maximizing the average number of scheduled processes. Nevertheless, 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 [7]. Consequently, several mechanisms have been proposed in recent years to provide RT support in the Linux Kernel (e.g, RTLinux [8], Low-Latency patch [9], PREEMPT_RT [10] patch), opening up the possibility of its use for RT systems [6].

Containers are considered a lightweight virtualization approach as containers do not deploy any guest OS. Instead, while the host OS adopts a RT Kernel supporting preemption and RT Scheduling mechanisms, the containers link their binaries and libraries to the host's RT OS.

B. Sharing computing resources with RT processes

Containers use the Kernel's features Cgroups to isolate resources in a multi-tenant environment. Cgroups capabilities to control and limit resources allow defining CPU sharing policies for collocated containers. For instance, the work in [11] proposes a CPU-allocation mechanism that dynamically adjusts the number of CPUs allocated to containers according to each container CPU-time demand and the system's load. To define the CPU limits for each container, authors use Cgroups subsystems *cpu.quota* and *cpu.period* which are only visible by the non-RT Scheduler Completely Fair Scheduler (CFS). A similar mechanism that uses these two Cgroups subsystems is presented in [12] aiming to allocate 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, we consider a MEC system deploying the Linux RT-Kernel to provide RT guarantees to RT containers. In addition, we propose a CPU sharing mechanism that allows non-RT containers to exploit underutilized CPUs allocated to RT containers. To do so, we leverage Cgroups subsystems **cpuset** [1] and **CPU-shares** [2]. While the former allows defining the set of CPUs that a container is allowed to use when running its applications, the latter defines the relative amount of CPU-time that the container is allowed to use.

III. System model

This paper considers a MEC server consisting of M CPUs. 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:

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

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

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

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 $\sum_{m=1}^{M} I_{rm} \geq C_r$. By orthogonal allocation we refer to the fact that two RT containers can not be allocated the same CPU, such that 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 and opportunistically try to use the underutilized CPUs assigned to RT containers. While CPU allocation decisions for non-RT containers can change over time, such decisions should meet the CPU requirement $C_n \leq M$ for each non-RT container $n \in \{L_2 \cup L_3\}$. Note that the CPU allocation here refers to defining the set of CPUs that containers either RT or non-RT are allowed to use. Then, the host RT-Kernel actually allocates CPU-time to containers on specified CPUs, according to the Kernel scheduling policy and the priority of the container's instantiated application.

[1] https://www.kernel.org/doc/Documentation/cgroup-v1/cpusets.txt
[2] https://man7.org/linux/man-pages/man7/cgroups.7.html

To compute parameters, this model assumes a slotted time $t \in \{0, 1, 2, \ldots\}$. The time-slot here, though, differs from the MEC's cycle duration in that this time-slot 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 time-slot. Because the time-slot granularity is arbitrary 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 system 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 time-slots.

As stated, temporal spikes of measured metric samples would affect further model computation. For this reason, this paper computes model parameters (e.g., per-processor CPU utilization and container's CPU usage) through a variation of the common Exponential Moving Average (EMA) [13]. Rather than computing such parameters based on the snapshot of the last sample in a time-slot, EMA provides smooth predicted samples for a general time series aiming to capture the trend from previous data. To illustrate how this model computes EMA, consider a time-slot of arbitrary granularity and a time slicing window of size $W_{\text{[time-slots]}}$. 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 time-slots. 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 $x(t)$ the EMA is computed as follows:

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

where $\alpha = \frac{2}{W+1}$ is the smoothing constant which assigns the greatest weight to the contribution of previous samples as given by $SMA(X(t))$. Unlike common EMA implementations which keep the predicted EMA as the contribution from previous data on subsequent computation of new samples, this approach does not save any EMA value. Rather, this approach keeps previous measured samples (not the predicted ones) in the ring buffer $X(t)$. Those samples are then used as the contribution of past data as in SMA to compute the EMA. In this case, the value obtained after the EMA calculation is further used to compute parameters. By the end of time-slot 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 time-slot [14]. Measuring the CPU utilization every time slot, this model buffers the previous W CPU utilization measurements.

For each CPU $m \in \{1, 2, \ldots, M\}$, define the virtual ring buffer $U_m(t)$ containing the previous W samples of CPU utilization. Updated every time-slot, 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 time-slot $(t-w)$, 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:

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

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 time-slot t, given by:

$$G_m(t) = 1 - U_m^*(t) \quad (3)$$

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 $n \in \{L_1 \cup L_2 \cup L_3\}$, define the virtual ring buffer $Q_n(t)$. Updated every time-slot, 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 time-slot $(t-w)$, where $w \in \{1, 2, \ldots, W\}$.

Denoted by $Q_n^*(t)$, this model computes the CPU usage of container n in time-slot 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:

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

IV. Opportunistic CPU sharing

Although allocating orthogonal CPUs to RT containers reduces resource contention latency caused by collocated RT processes, orthogonal CPU allocation increases CPU underutilization. This section presents a CPU sharing mechanism that allows non-RT containers to exploit underutilized CPUs pre-allocated to RT containers.

A. CPU and CPU-shares Allocation problem

Consider a system controller that monitors and computes the per CPU utilization $U_m^*(t)$ following (2), 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 container $n' \in \{L_1, L_2, L_3\}$. Furthermore, let $\mu_n(t) = (\mu_{n1}(t), \mu_{n2}(t), \ldots, \mu_{nM}(t))$ represent the CPU allocation vector for the non-RT container $n \in \{L_2, L_3\}$. Here,

$\mu_{nm}(t) \in \{0,1\}$ denotes the CPU m allocation decision, such that $\mu_{nm}(t) = 1$ if the CPU m is allocated to container n, and $\mu_{nm}(t) = 0$ otherwise. Based on the current $U^*_m(t)$ and $Q_n*(t)$, the network controller chooses the CPU allocation vector to each non-RT container n subject to its CPU requirement C_n, every time-slot t.

Because RT containers run RT applications which are sensitive to processing interference from collocated workloads, the system controller should also define a CPU-time policy which prioritizes RT containers from CPU-time allocation by the host's RT-Kernel. Not only does the system controller allocates CPUs to non-RT containers, but the system controller also decides on the relative amount of CPU-time that each non-RT container is allowed to use (referred to here as CPU-shares) on the allocated CPUs. Thus, the goal is to design an algorithm that allocates CPUs and CPU-shares to non-RT containers that solves the following optimization problem:

$$\text{Maximize:} \sum_{n\in\{L_2\cup L_3\}} \sum_{m=1}^{M} \omega_{nm}(t)\mu_{nm}(t)$$

$$\text{subject to:} \sum_{m=1}^{M} \mu_{nm}(t) \geq C_n, n \in \{L_2 \cup L_3\}$$

$$\sum_{n\in\{L_2\cup L_3\}} S_{nm}(t)\mu_{nm}(t) + \sum_{r\in L_1} S_{rm}(t)I_{rm} \leq 1,$$

$$m \in \{1,2,...,M\}$$

$$\mu_{nm}(t) \in \{0,1\}, \forall n,m$$

$$0 \leq S_n(t) \leq 1$$

This linear problem seeks to maximize a weighted sum of CPU allocation subject to each container's CPU requirement and CPU-shares limits, where $\omega_{nm}(t)$ denotes a positive weight for the non-RT container n when using CPU m. On the other hand, $S_{nm}(t)$ represents the decision on the relative amount of CPU-time that container n is allowed to use. Note that the CPU-shares of collocated containers sharing the same CPU can not sum more than one.

The above problem reduces to a generalized Maximum Weight Match (MWM) problem where the weight of a pair (n,m) is given by $\omega_{nm}(t)$. Described below is PRINCIPIA, a constant factor approximation algorithm to solve the problem in (IV-A). It allocates CPUs to non-RT containers based on the per processor CPU availability and the container's CPU usage.

B. PRINCIPIA: opportunistic CPU sharing algorithm

PRINCIPIA is a constant factor approximation algorithm to solve the MWM problem in (IV-A). Solving this problem requires solving the MWM problem on a $|L_2 \cup L_3| \times M$ bipartite graph of $|L_2 \cup L_3|$ non-RT containers and M CPUs. Figure 1 illustrates a bipartite graph where PRINCIPIA allocates CPUs to non-RT containers. Here, we assume that the pool of CPUs have been pre-allocated to RT containers and non-RT opportunistically try to use those CPUs.

Let $G(V,E)$ denote the bipartite graph in figure 1 where the vertex set V is decoupled into the set of non-RT containers $V_1 = \{L_2 \cup L_3\}$ 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 $\omega_{nm}(t)$. Let $\rho_m(t)$ denote the number of containers for which CPU m has been allocated until the beginning of time-slot t. Because each CPU m has been pre-allocated to a given RT container, $\rho_m(t) \geq 1$.

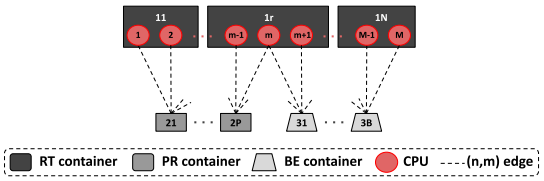


Fig. 1: Example MEC system showing containers with different priority (e.g., RT, PR, and BE). For each non-RT container and each CPU there exists an edge (n,m) whose weight defines the basis for opportunistic CPU allocation.

Computing probable CPU and CPU-shares allocation: Inspired by the Law of Attraction, PRINCIPIA defines the weight $\omega_{nm}(t)$ as the attraction of a non-RT container n to use the CPU m. To compute $\omega_{nm}(t)$, the per processor CPU availability $G_m(t)$ represents the "mass" of CPU m. Similarly, the container's CPU usage $Q^*_n(t)$ represents the "mass" of the non-RT container n. Moreover, $\rho_m(t)$ represents the "distance" between the CPU m and a given container n. Thus, the attraction of a non-RT container n to use the CPU m is computed as follows:

$$\omega_{nm}(t) = \frac{G_m(t)Q^*_n(t)}{\rho_m(t)^2} \tag{5}$$

Here, the attraction of container n to use CPU m varies proportionally to both the CPU availability and the container's CPU usage, and varies inversely as the number of allocated containers to that CPU. Thus, the attraction of container n to each CPU m is contained in the vector $\Omega_n = (\omega_{n1}(t), \omega_{n2}(t), \ldots, \omega_{nM}(t))$.

PRINCIPIA also computes the CPU-shares, i.e., the relative amount of CPU-time that each container is allowed to use. Let υ_k denote the control constant associated with each priority policy $k \in \{1,2,3\}$, such that $\upsilon_1 < \upsilon_2 < \upsilon_3$. Also, define $s_n(t)$ as the weight of container n on the CPU-shares allocation among containers sharing the same CPU. Inspired by the common Inverse-Square Law, $s_n(t)$ represents CPU intensity of container n which is computed as follows:

$$s_n(t) = \frac{Q^*_n(t)}{\upsilon_k^2} \tag{6}$$

Here, $s_n(t)$ varies proportionally to the container's CPU usage $Q^*_n(t)$, and varies inversely as the square of the control constant υ_k. 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 PR container is twice to 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 to that of RT containers.

Greedy Maximal Weight Match solution: PRINCIPIA proposes a Greedy Maximal Weight Match (P_GMWM) heuristic to solve the MWM problem in (IV-A). The P_GMWM aims to provide a greedy and on-line solution which can be computed with less overhead than solving the MWM.

The P_GMWM consists of finding the maximal match for each non-RT container $n \in \{L_2 \cup L_3\}$. A maximal match is defined as the subset $E_n \in E$ containing the C_n (e.g., CPU requirement of container n) edges (n, m) with the largest weights in $\Omega_n(t)$, such that $E_n(t) = \{(n, i), (n, i')|\omega_{ni}(t) > \omega_{ni'}(t)\}$, where $i, i' \in \{1, 2, \ldots, M\}$, $i \neq i'$, and $|E_n(t)| = C_n$.

Let $\mu_n^*(t) = (\mu_{n1}^*(t), \mu_{n2}^*(t), \ldots, \mu_{nM}^*(t))$ be the solution vector to the MWM problem in (IV-A) following the P_GMWM, where $\mu_{nm}^*(t)$ is given by:

$$\mu_{nm}^*(t) = \begin{cases} 1, & \text{if } (n, m) \in E_n(t) \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

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:

$$S_{nm}^*(t) = \frac{s_n(t)}{\sum_{n \in \{L_2 \cup L_3\}} s_n(t)\mu_{nm}^*(t) + \sum_{r \in L_1} s_r^*(t) I_{rm}} \tag{8}$$

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 from avoiding CPU starvation, the CPU "intensity" prioritizes RT containers on the CPU-shares allocation.

How does the control constant υ_k enable priority based CPU-shares allocation to RT containers in contention when several containers share the same CPU? As stated, the CPU "intensity" of non-RT containers decreases as their control constants get greater than the RT one i.e., υ_1. As a result, computing the container's CPU intensity based on the inverse square of the control constant, enables controlling the influence of a given non-RT container on the allocation of CPU-shares. Put another way, the greater the control constant of non-RT containers, the more priority the RT container receives for the CPU-shares allocation.

V. Evaluation

We consider a MEC system [3] consisting of four CPUs, i.e. $M = 4$, as depicted in figure 2. Using Linux Containers (LXC), this system deploys a combination of two RT containers, two PR containers, and two BE containers.

The methodology consists of evaluating the processing latency of deployed RT containers while sharing computing resources with collocated non-RT containers. Let $L_1 = \{11, 12\}$ be the set of RT containers whose CPU requirements are assumed as $C_{11} = C_{12} = 2$. Similarly, let $L_2 = \{21, 22\}$ be the set of PR containers whose CPU requirements are assumed as $C_{21} = C_{22} = 2$. Finally, let $L_3 = \{31, 32\}$ be 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 non-RT containers use CPUs assigned to RT containers as scheduled by the RT-Kernel; (ii) PRINCIPIA, where the PRINCIPIA mechanism defines on which CPUs non-RT containers should be scheduled by the RT-Kernel, and controls the amount of CPU-time that non-RT containers get granted on those CPUs.

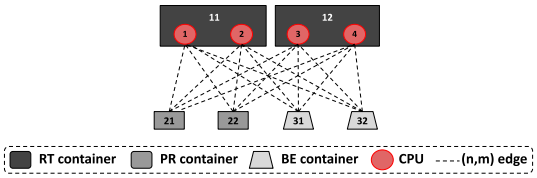


Fig. 2: CPU allocation scenario. RT containers are allocated orthogonal CPUs. Both PR and BE containers receives CPU-time from any system's CPU according to scheduling policies.

To emulate an application's workload, non-RT containers run different instances of the synthetic benchmark tool *stress-ng* *stress-ng*[4] 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 which performs 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 mmap(2)/munmap(2)[5] (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. Finally, each RT container runs a stressors with RT priority 99 (i.e., by setting the Linux RT attribute chrt = 99). The stressors run as a single thread process instructed to rapidly change the CPU affinity. Switching this process's CPU affinity on the evaluated CPUs enables emulating a scenario where the RT-Kernel Schedules multiple processes on the evaluated CPU's run queues.

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[6] provides an estimate of the system's RT latency by measuring the difference between the

[3]General purpose server equipped with eight Intel i7-8750H physical processors at 2.20 GHz, and 32 GiB of memory. The MEC's OS is the Ubuntu 20.04 with low-latency Linux Kernel version 5.4.0.125.126.

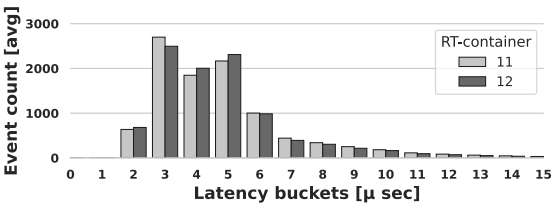
[4]https://wiki.ubuntu.com/Kernel/Reference/stress-ng

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

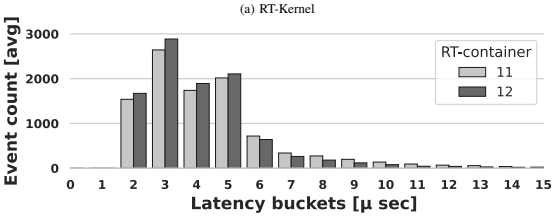
[6]https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start

time at which the thread signals to wake up and the actually wake up time. Here, each experiment captures the processing latency as reported by the cyclictest during a time span of 5 minutes.

Figure 3 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.



(a) RT-Kernel



(b) PRINCIPIA

Fig. 3: 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.

These distributions show how PRINCIPIA mitigates the impact on the processing latency. For instance, on average, events on the latency bucket 2 μsec are more than 100% greater in PRINCIPIA than in the RT-Kernel. Not only these benefits are evidenced as relative frequency increase of events on lower latency buckets, but also these distributions tails. Providing hints of the WCET of RT containers, figure 4 shows the increase in percentage of tail latency events when using the RT-Kernel in comparison with PRINCIPIA.

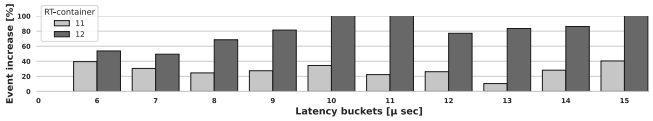
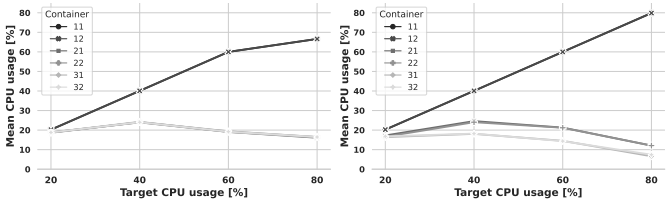


Fig. 4: Processing latency increase when using the RT-Kernel in comparison with PRINCIPIA.

Although the PRINCIPIA mechanism focuses on providing a CPU sharing policy, dynamically controlling how non-RT container use allocated CPUs to RT containers mitigates the impact on their processing latency by reducing the WCET events, in some cases by much as 100% in comparison with the RT-Kernel.

Could CPU intensive RT containers starve collocated non-RT containers when using PRINCIPIA? To answer this question, figure 5 shows the mean CPU usage measured at each deployed container as a function of their target CPU usage. Each container generates CPU load by running as many CPU stressors of the *stress-ng* tool as its CPU requirement (e.g., two CPU stressors per container). Because RT containers are intended to host RT applications, RT containers run their CPU stressors with the RT priority chrt = 95. Conducted experiments consist of measuring each container's CPU usage, while varying the target CPU usage of each CPU stressors running on each container. A set of twelve experiments, whose duration is 5 minutes, is conducted per target CPU usage.



(a) RT-Kernel (b) PRINCIPIA

Fig. 5: Mean CPU usage [%] as measured by each container $n \in \{L_1 \cup L_2 \cup L_3\}$.

The CPU stressors instantiated on RT-containers can preempt the CPU stressors instantiated on non-RT containers. As a result, RT containers reach 100% processing throughput in the vast majority of cases with the exception of RT-Kernel scenario for an overloaded system (i.e., 80% target CPU usage), where the measured CPU drops by 18% in comparison with PRINCIPIA. The reason is that PRINCIPIA CPU sharing mechanism controls and prioritizes CPU-time allocation to RT containers.

Not only PRINCIPIA provides prioritized access to RT-containers, but also provides differentiated service to PR containers. As shown in figure 5b, PR containers perceive higher CPU usage than BE containers despite how overloaded the system is. Using the control constant υ_k, PRINCIPIA can control the influence of non-RT containers on the CPU-shares allocation, either to provide differentiated allocation, or as conservative mechanism protecting RT containers from potential processing interference from collocated workloads.

VI. Conclusions

This paper models CPU sharing in MEC servers deploying applications with diverse execution time requirements. Also, this paper proposes PRINCIPIA, an opportunistic CPU sharing mechanism for containerized virtualization in MEC hosting multiple applications with heterogeneous execution time requirements. PRINCIPIA opportunistically allocates CPUs to non-RT containers exploiting available CPU-time from CPUs pre-allocated to RT-containers. Using a control constant based on the containers classification policy, PRINCIPIA controls and limits the relative amount of CPU-time that each container is allowed to use in scenarios of CPU contention.

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.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] S. Xi, C. Li, C. Lu, C. D. Gill, M. Xu, L. T. Phan, I. Lee, and O. Sokolsky, "Rt-open stack: Cpu resource management for real-time cloud computing," in *2015 IEEE 8th International Conference on Cloud Computing*, 2015, pp. 179–186.

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

[6] 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.

[7] 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

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

[9] 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

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

[11] 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.

[12] 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.

[13] 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.

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