QoS-Aware Scheduling of Remote Rendering for Interactive Multimedia Applications in Edge Computing

Ruitao Xie, Junhong Fang, Junmei Yao, Kai Liu, *Senior Member, IEEE*, Xiaohua Jia, *Fellow, IEEE*, and Kaishun Wu

Abstract—Leveraging emerging edge computing and 5G networks, researchers proposed to offload the 3D rendering of interactive multimedia applications (e.g., virtual reality and cloud gaming) onto edge servers. For high resource utilization, multiple rendering tasks run in the same GPU server and compete against each other for the computation resource. Each task has its requirement for performance, i.e., QoS target. A significant problem is how to schedule tasks so that each preset QoS is met and the performance of all tasks are maximized. We make the following contributions. First, we formulate the problem into a QoS constrained max-min utility problem. Second, we find that using the common natural logarithm as a utility function overly promotes one performance but demotes another. To avoid this phenomenon, we design a special utility function. Third, we propose an efficient scheduling algorithm, consisting of a resolution adjustment algorithm and a frame rate fair scheduling algorithm, both of which interact with each other. The former selects resolutions for tasks and the latter decides which task to process. We evaluate our method with actual rendering data, and the simulations demonstrate that our method can effectively improve task performance as well as satisfy QoS simultaneously.

Index Terms—Edge computing, task scheduling, remote rendering

1 Introduction

Edge computing emerges as localized clouds [1], [2]. Due to the proximity to users, it can achieve a low response time. Cloud-based interactive multimedia applications, such as virtual reality and cloud gaming [3], leverage cloud resources to process their computation-intensive workloads, so as to remove powerful and expensive hardware from user devices and lead to lightweight clients. Pioneering commercial applications include OnLive [4], GeForce Now [5], CloudXR [6] *etc.* However, these interactive applications need high-throughput and low-latency internet connections, which are challenging for users due to the long distance from data centers. A promising solution to overcome this is leveraging emerging mobile edge computing. More specifically, edge-assisted interactive applications offload computation-intensive 3D rendering onto GPU-based infrastructures in mobile edge computing and stream edge-rendered visuals to end users over 5G connections, as proposed in [7], [8]. As such, the proximity to end users can greatly reduce latency.

As shown in Fig. 1, an edge-assisted interactive application is composed of three parts distributed in different locations: the application logic running in a cloud server, the rendering engine running in an edge server, and the display and control component running in a user device. The three components interact with each other. Specifically, the rendering engine receives rendering commands from the application logic, executes them, and transfers the rendered frames to the user device. The user device generates control commands, transfer them back to the cloud server and let it update the application logic.

For edge-assisted interactive applications, an edge computing provider supplies infrastructure resources and rendering services to users. The provider makes a service level agreement (SLA) with a user, specifying quality of service (QoS) targets and economical penalties associated with SLA violations. For interactive applications, the performance which concerns us includes resolution, frame interval/rate, and latency. We set QoS targets by defining requirements for each performance.

- *Ruitao Xie, Junhong Fang, and Junmei Yao are with the College of Computer Science and Software Engineering, Shenzhen University, Shenzhen 518060, China. E-mail: {drtxie, fangjh111b}@gmail.com, yaojunmei@szu.edu.cn.*
- *Kai Liu is with the College of Computer Science, Chongqing University, Chongqing 400044, China. E-mail: liukai0807@cqu.edu.cn.*
- *Xiaohua Jia is with the Department of Computer Science, City University of Hong Kong, Hong Kong, China. E-mail: csjia@cityu.edu.hk.*
- *Kaishun Wu is with the College of Computer Science and Software Engineering, Shenzhen University, Shenzhen 518060, China, and also with the PCL Research Center of Networks and Communications, Peng Cheng Laboratory, Shenzhen 518066, China. E-mail: wu@szu.edu.cn.*

Manuscript received 26 Nov. 2021; revised 24 Apr. 2022; accepted 28 Apr. 2022. Date of publication 3 May 2022; date of current version 26 July 2022.
This work was supported in part by China NSFC under Grants 61802263, 62072317, and 62172064, in part by the Grant from Research Grants Council of Hong Kong under Grant CityU 11202419, in part by the Faculty Research Fund of Shenzhen University under Grant 860/000002110325, in part by the China NSFC under Grants U2001207 and 61872248, in part by Guangdong NSF under Grant 2017A030312008, in part by Shenzhen Science and Technology Foundation under Grants ZDSYS20190902092853047 and R2020A045, in part by the Project of DEGP under Grants 2019KCXTD005 and 2021ZDZX1068, and in part by Guangdong "Pearl River Talent Recruitment Program" under Grant 2019ZT08X603.
(Corresponding author: Kai Liu.)
Recommended for acceptance by Y. Yang.
Digital Object Identifier no. 10.1109/TPDS.2022.3172121

1045-9219 © 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: SHENZHEN UNIVERSITY. Downloaded on August 09,2022 at 09:18:04 UTC from IEEE Xplore. Restrictions apply.

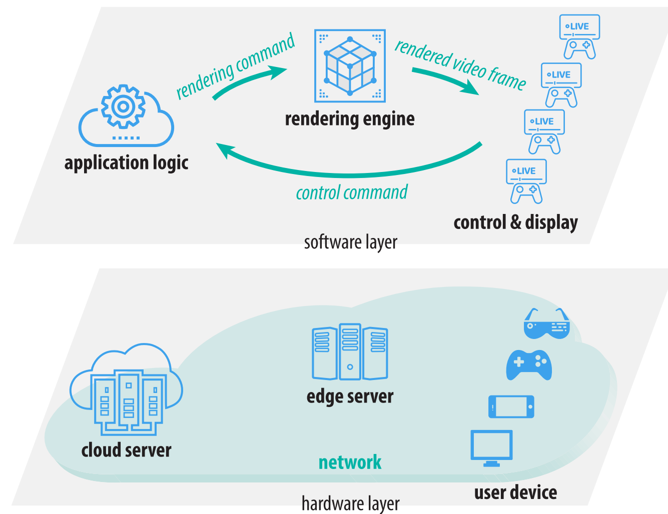


Fig. 1. The architecture of edge 3D rendering.

To improve resource utilization, multiple rendering tasks coexist on an edge server and share a common processor, each supplying a rendering service for one interactive application. They compete against each other for the computation resource. Thus, a *QoS-aware rendering task scheduling problem* arises, that is how to schedule rendering tasks so that all tasks could receive good performance and fulfill QoS targets at the same time.

Given a set of rendering tasks assigned on a server and a set of resolutions to use, when the server is available, two decisions have to be made in scheduling: 1) which task to schedule; 2) which resolution to use for rendering. Making these decisions is complicated, with the ultimate goal being to deliver the best possible rendered frames to each user promptly. For one thing, user requirements must be met; for another, if abundant resource is available, we need to make good use of that to maximize user satisfaction.

Naively scheduling in a round robin manner using fixed resolutions leads to a fixed frame rate for all tasks, as they will be executed at the same frequency. This may achieve neither of the objectives mentioned above, because different users may have different frame rate requirements, and it cannot make good use of abundant computation resource as well. Furthermore, promoting resolutions increases both processing time and transmission time, and aggressively promoting may violate latency requirements. As such, the scheduling algorithm must consider the trade-offs between each pair of performance and try to make optimal choices. Different from traditional scheduling problems, the biggest challenge in our problem is to achieve a balance between multiple distinctive performance metrics, under hard constraints over resolution, frame interval, and latency that guarantee interactivity.

Overall, in this paper we make the following contributions:

- We formulate the QoS-aware rendering task scheduling problem into a QoS constrained max-min utility problem.
- With the natural logarithm as a utility function, we find that it overly promotes one performance but demotes another. To achieve multiple QoS targets at the same time, we design a special utility function.
- We propose an algorithm to address the scheduling problem. It is composed of a resolution adjustment (RA) algorithm and a frame rate fair (FRF) scheduling algorithm, both of which interact with each other. The former selects resolutions for tasks intelligently, and the latter decides which task to process.

To evaluate our method practically, we produce simulation environments using actual rendering data. We compare our method with several classical scheduling algorithms. Our simulations demonstrate that our method performs better than the others under various computing loads. It can effectively improve task performance and satisfy QoS targets simultaneously.

The rest of this paper is organized as follows. Section 2 introduces the related works. Section 3 presents an overview of edge 3D rendering. Section 4 discusses quality of service and task assignment. Section 5 presents the formulation of the QoS-aware rendering task scheduling problem. Section 6 presents our algorithm. In Section 7, we introduce simulation setup. In Section 8, we present simulation and performance evaluation. Section 9 concludes the paper.

2 Related Work

We review the existing works related to our problem, and categorize them into two groups as follows.

2.1 Resource Management

This is an active research field in resource-sharing systems. Existing works [9], [10] pack tasks to machines based on their requirements of all resource types. Li *et al.* in [11] proposed an algorithm for allocating requests to servers in order to minimize response time and maximize the profit of service providers. Li *et al.* in [12], [13] investigated workflow scheduling problems in cloud systems. Ren *et al.* in [14] proposed a dependent task offloading framework for multiple mobile applications, and in [15] they studied a distributed computation offloading problem with delay constraints for edge computing. Cao *et al.* in [16] optimized response time of interactive mobile applications using intelligent computation offloading. These task assignment/offloading problems allocate a given set of tasks (dependent or not) onto a set of cloud/edge nodes to minimize job completion time, minimize resource consumption or maximize profit. After task assignment/offloading, multiple tasks may coexist on a server and share the same processor (GPU/CPU). How to schedule such concurrent tasks is addressed in this paper. Some works propose algorithms to schedule multiple applications to share the same GPU to improve GPU utilization [17]. However, they do not consider the user-defined performance requirements for individual applications, like the QoS requirements in our work. Zhang *et al.* in [18] proposed a task assignment system GCloud to assign cloud gaming tasks to machines. Zhang *et al.* in [19], [20] proposed virtualized GPU scheduling algorithms to meet the required framerate and maximize GPU usage for cloud gaming. They do not consider resolution optimization and the impact of network transmission as in our work. Scheduling data processing jobs on distributed compute clusters is another active research field, whose goal is usually to minimize average job completion time [21], [22]. Our work maximizes user

Authorized licensed use limited to: SHENZHEN UNIVERSITY. Downloaded on August 09,2022 at 09:18:04 UTC from IEEE Xplore. Restrictions apply.

satisfaction, which includes frame rate and resolution, and minimizing job completion time only affects frame rate.

2.2 Remote Rendering

Several commercial cloud gaming platforms have launched recently, such as OnLive [4], GeForce Now [5], Stadia [23], CloudXR [6] *etc.* These services need high-throughput and low-latency internet connections, which is hard to ensure. A promising solution is to leverage emerging edge computing close to users. Researchers have proposed some prototype systems, such as the edge-assisted VR gaming in [24] and the AR application in [25]. However, they did not consider frame-by-frame GPU scheduling like what we proposed. For edge-assisted or cloud-assisted interactive applications, to improve the quality of experience (QoE) of users and reduce latency, lots of techniques have been proposed from different perspectives. Liu *et al.* in [26] proposed a parallel rendering and streaming mechanism to reduce the streaming latency, and a remote VSync driven rendering technique to minimize display latency in remote rendering. Liao *et al.* in [27] proposed to use compression techniques to save bandwidth for transmitting rendering commands and geometry data. Zhang *et al.* in [7] proposed a bitrate control algorithm which determines the compression parameters according to network dynamics. Different from these works, we improve performance by determining frame rate and resolution before rendering and encoding.

3 System Model

Edge 3D rendering plays an important role in emerging interactive applications, such as AR/VR and video gaming. As shown in Fig. 1, a system may consist of cloud servers, edge servers and user devices. Cloud servers host the core logic of an application; edge servers provide rendering and encoding; user devices provide decoding and displaying. An information loop forms among them. Cloud servers generate rendering commands (which manipulate geometric objects to generate video scenes as discussed in [27]) and transfer them to edge servers; edge servers produce frames and transfer them to user devices; user devices generate control commands, transfer them backward to cloud servers and let the latter update the application logic. It is noted that the application logic may be offloaded to edge servers as well. The rendering task scheduling problem is the same in that situation.

Interactive applications are greatly different from traditional ones like video streaming. In the latter, there is no interaction between a user and a video, and the user is interested in all the frames. Thus, the rendering engine queues and processes all the rendering commands, and the frames can be cached on the user device if not viewed immediately. However, in the former, an application has an internal state. The user actively changes the state by providing input, and immediately observes the state via rendered frames. The user is only interested in the latest state resulting from the latest input, and previous states can be discarded if their frames cannot be rendered in time. As a result, the engine only processes the latest rendering command which overrides any previous unprocessed commands. In this paper, we only focus on the interactive case.

For edge-assisted interactive applications, an edge computing provider supplies infrastructure resources and rendering services to users. When a user plans to offload its rendering task, it states resource demand. Then, the user and the provider make a service level agreement on the quality of service (QoS). If the agreed QoS is not met, then the customer may ask for a refund from the provider.

For a rendering task, once its resource demand and QoS are set, the provider will assign it to an edge server. To improve the resource utilization of servers, multiple tasks may coexist on a server. The running of each task is orchestrated by a scheduler to achieve each preset QoS. We will discuss QoS and task assignment in the following section. The scheduling problem will be formulated in Section 5.

4 QoS and Task Assignment

4.1 Quality of Service

We first discuss the QoS for interactive applications. Resolution and frame rate are two common key concerns of video applications. Thus, we consider them in QoS. High resolution leads to better image quality, and a high frame rate is required for smooth user interaction. Furthermore, latency is critical for the user experiences of interactive applications, such as AR and VR [28], [29]. Low latency is required to ensure responsiveness for user interactions. In this paper, we only consider the latency relevant to rendering scheduling, that is the latency from the time when a rendering command arrives at the system to the time when a user receives the associated rendered frame. These factors, resolution, frame rate, and latency, play important roles in interactive applications as discussed in [7].

There is a trade-off between resolution and latency. High resolution leads to long rendering time and transmission time, both resulting in long latency. Also, there is a trade-off between resolution and frame rate. High resolution leads to low-frequency scheduling and thus a low frame rate.

The requirements for these performance (resolution, frame rate, and latency) constitute a set of QoS targets. The trade-offs mentioned above should be considered in defining a set of QoS targets. Otherwise, the requirements may be unmet. A problem arises, that is how to define QoS targets. An in-depth investigation of this problem depends on implementation details of the system and pricing model [30], [31]. It is independent of the rendering scheduling problem and thus is beyond the scope of this paper.

4.2 Task Assignment

For a task, given its demand for resources, such as network bandwidth, GPU/CPU memory, GPU/CPU type, GPU/CPU amount, storage *etc.*, the task will be assigned on a server with sufficient resources such that its demand is met. To improve the resource utilization of edge servers, multiple tasks may be assigned on a server, if the total demand does not exceed the server capacity. The rendering task assignment problem can be formulated as a multi-dimensional bin packing problem. Existing algorithms [9], [10], [18] can solve the problem. However, multiple rendering tasks may share resources and assigning them together can reduce resource consumption. In our previous work [32], we considered resource sharing among rendering tasks and

Authorized licensed use limited to: SHENZHEN UNIVERSITY. Downloaded on August 09,2022 at 09:18:04 UTC from IEEE Xplore. Restrictions apply.

proposed a sharing-aware offloading algorithm to reduce resource consumption.

5 Formulation of Task Scheduling

For a task, we introduce a utility function to quantify its performance. Since there are multiple tasks to optimize, we maximize the minimum utility among them. The rationale of this max-min formulation is to try our best to satisfy the requirements of all users and provides extra performance (e.g., better frame quality) if computation resource is abundant.

Here we give a quick overview of the task scheduling formulation, which will then be detailed in the following sections. Given n rendering tasks $\{s_1, s_2 \dots s_n\}$, each task is constantly receiving rendering commands from the servers that host the application logic (see Fig. 1), and denote o_{ik} as the kth command received by s_i. The execution of o_{ik} constitutes a rendering command execution task, aka RCE-task, denoted as $\hat{o}_{ik}$, which when finished will result in an image sent to the user. Note that a rendering task only caches the latest command, so if the execution of o_{ik} does not begin before $o_{i,k+1}$ arrives, o_{ik} will be discarded. This is due to the nature of interactive applications where users always want to see the latest response to their input, and if previous responses are shown then the application will appear lagging. If the RCE-tasks belonging to s_i are executed very frequently in high resolution, then the user served by s_i will experience high frame rate and best image quality, which is desirable. However, it is at the cost of the experience of other users due to the limited computation resources. Our goal is to schedule and execute the RCE-tasks $\{\hat{o}_{ik}\}$ so that all QoS requirements (resolution, frame rate and latency) are satisfied and the utility function that measures user experience is maximized.

Formally, we would like to find an order to execute the RCE-tasks, $\hat{O} = [\hat{o}_{i_1k_1}, \hat{o}_{i_2k_2}, \dots, \hat{o}_{i_Nk_N}]$, where $\hat{o}_{i_tk_t}$ is the RCE-task for the k_tth command of the rendering task s_{k_t}. To execute the RCE-tasks, We also need to select a resolution $r_{i_tk_t}$ for each RCE-task $\hat{o}_{i_tk_t}$. Executing all the RCE-tasks in the given order using the selected resolutions will lead to different average resolution, frame rate and latency for different rendering tasks, and these performance metrics are combined into a utility function, optimized under constraints specified by QoS requirements.

5.1 QoS Constrained Max-Min Utility Problem

In this section, we first define the utility function for a rendering task and then present the formulation that maximizes the total utility of all rendering tasks. For a rendering task, given its required resolution $r_{\min}$ and the achieved one r (that is, the rendering commands of this task are executed using resolution r), we formulate its performance as $u(\frac{r}{r_{\min}})$, where $u(x)$ is a concave nondecreasing utility function. It is positive if x is larger than one and negative otherwise. Similarly, for the maximum tolerable latency $d_{\max}$, we formulate the performance to be $u(\frac{d_{\max}}{d})$. However, for the minimum required frame rate $f_{\min}$, we use its reciprocal, namely, frame interval $h_{\max} = \frac{1}{f_{\min}}$, to measure the performance. Frame interval is the time elapsed between two consecutive frames, and frame rate is a statistic over a period of time. The former is a stronger requirement, in that a constant frame interval guarantees an even distribution of received frames over time but a constant frame rate does not. So for the frame rate requirement, the performance measurement is $u(\frac{h_{\max}}{h})$, where h is the measured frame interval. The overall performance as a weighted sum

$$u(r, h, d) = \theta_r u\left(\frac{r}{r_{\min}}\right) + \theta_h u\left(\frac{h_{\max}}{h}\right) + \theta_d u\left(\frac{d_{\max}}{d}\right), \quad (1)$$

where θ_r, θ_h and θ_d are the weights for balancing each kind of performance. They are all nonnegative and summed to 1.

We now discuss how to compute the utility for a rendering task after all of its rendering commands have been executed. Given the order of RCE-tasks $\hat{O} = [\hat{o}_{i_1k_1}, \dots, \hat{o}_{i_Nk_N}]$, we can execute them one by one. After each execution, a frame is produced, which is then encoded and transmitted. However, during encoding or transmission, congestion may happen and the frame may be dropped, so the frame is not received by the user and does not contribute to the user experience (and therefore the utility). To focus only on the effective RCE-tasks, for a rendering task s_i, we define a subsequence of $\hat{O}$ whose resulted frames are all received by the user as $\hat{O}_i = [\hat{o}_{i_tk_t} \in \hat{O} | i_t = i \wedge \rho_{i_tk_t} = 1]$, where $\rho_{i_tk_t} = 1$ if the frame obtained by executing $\hat{o}_{i_tk_t}$ is successfully received by the user, and 0 otherwise. For simplicity we redefine the indices of $\hat{O}_i$ and denote it by $\hat{O}_i = [\hat{o}_{i1}, \hat{o}_{i2}, \dots, \hat{o}_{i,m_i}]$, where $\hat{o}_{ij}$ represents the RCE-task for the jth *executed* command of task s_i and whose frame is *received* by the user. We take these user-received frames to evaluate performance. To execute $\hat{o}_{ij}$, a resolution r_{ij} is selected, and after its execution, frame interval h_{ij} and latency d_{ij} are obtained. An instant utility can be got from (1), that is $u(r_{ij}, h_{ij}, d_{ij})$. The utility of a task is defined as the average utility of all commands executed, denoted by u_i, that is

$$u_i = \frac{1}{m_i} \sum_{j=1}^{m_i} u(r_{ij}, h_{ij}, d_{ij}). \quad (2)$$

However, this utility function does not consider fluctuations in frame intervals which may lead to negative impact on user experience. To take that into account, we add a penalty term v_i related to the variance of the frame intervals, discussed in Section 5.3, into the utility function and define the final utility as follows:

$$\tilde{u}_i = u_i - \phi v_i, \quad (3)$$

where ϕ is a weight parameter. Note that u_i and $\tilde{u}_i$ are both utility functions, and by default we refer to $\tilde{u}_i$, unless stated otherwise.

Our problem is formulated as

$$\text{maximize} \quad \min_{i=1\dots n} \quad \tilde{u}_i \quad (4a)$$

$$\text{subject to} \quad \frac{1}{m_i} \sum_{j=1}^{m_i} r_{ij} \geq r^i_{\min} \qquad \forall i = 1 \dots n \quad (4b)$$

$$\frac{1}{m_i} \sum_{j=1}^{m_i} h_{ij} \leq h^i_{\max} \qquad \forall i = 1 \dots n \quad (4c)$$

$$\frac{1}{m_i} \sum_{j=1}^{m_i} d_{ij} \leq d^i_{\max} \qquad \forall i = 1 \dots n, \quad (4d)$$

Authorized licensed use limited to: SHENZHEN UNIVERSITY. Downloaded on August 09,2022 at 09:18:04 UTC from IEEE Xplore. Restrictions apply.

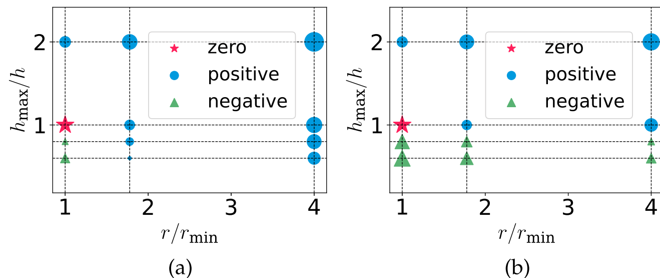


Fig. 2. The comparison between two forms of utility function, where $r_{\min}$ is 1920×1080 and $h_{\max}$ is $\frac{1}{30}$ s. Marker size is proportional to absolute value. (a) $\frac{1}{2}\log(\frac{r}{r_{\min}}) + \frac{1}{2}\log(\frac{h_{\max}}{h})$; (b) $\frac{1}{2}u(\frac{r}{r_{\min}}) + \frac{1}{2}u(\frac{h_{\max}}{h})$, where $u(x)$ is defined in (6) with $\alpha_r = \alpha_h = 1$.

where $r^i_{\min}$, $h^i_{\max}$, and $d^i_{\max}$ denote the required resolution, the maximum tolerable frame interval, and the maximum tolerable latency for task s_i respectively. The three constraints in the above formulation ensure the average performance of each kind. For a problem (4) and a scheduling policy, if all three constraints are satisfied, we say QoS satisfaction; otherwise, we say QoS violation. It is noted that we not only formulate latency in the objective but also limit it in the constraints. The reason is that for interactive applications which require immediate response to user input, a hard limit exists for latency.

5.2 Definition of *u(x)*

One might take the natural logarithm $\log(x)$ as utility function and formulate the overall performance as

$$\theta_r \log\left(\frac{r}{r_{\min}}\right) + \theta_h \log\left(\frac{h_{\max}}{h}\right) + \theta_d \log\left(\frac{d_{\max}}{d}\right). \tag{5}$$

However, this may cause an incentive to overly promote one performance but demote another. We take an example to illustrate this. We consider the objective which contains only two terms $\frac{1}{2}\log(\frac{r}{r_{\min}}) + \frac{1}{2}\log(\frac{h_{\max}}{h})$, where resolution r is chosen from {1920×1080, 2560×1440, 3840×2160} and frame interval h is chosen from {$\frac{1}{18}$ s, $\frac{1}{24}$ s, $\frac{1}{30}$ s, $\frac{1}{60}$ s}, corresponding to 18, 24 , 30 and 60 frame-per-second (FPS). Let $r_{\min}$ be 1920×1080 and the required frame rate be 30 FPS. We then illustrate all cases in Fig. 2a. It is shown that promoting resolution (to 2560×1440 or 3840×2160) but demoting frame rate (to 18 or 24 FPS) result in positive values, which is better than satisfying both.

In order to avoid the above issue, we must find a utility function such that $u(r,h,d) \geq 0$ only if all three kinds of performance requirements are satisfied, that is $\frac{r}{r_{\min}} \geq 1$, $\frac{h_{\max}}{h} \geq 1$, and $\frac{d_{\max}}{d} \geq 1$ are met simultaneously. To this end, we define a utility function as follows:

$$u(x) = \begin{cases} 1 - e^{1-x}, & x \geq 1, \\ \log(x) - \alpha, & x < 1, \end{cases} \tag{6}$$

where α is a nonnegative constant adjusted for penalizing. When $x > 1$, $u(x)$ is in $(0, 1]$; when $x = 1$, $u(x) = 0$; when $x < 1$, $u(x)$ is in $[-\infty, -\alpha)$. This is illustrated in Fig. 3.

This utility function has several advantages over $\log(x)$. First, it is upper bounded by one rather than increasing to infinity. This is helpful when balancing the trade-off among the three kinds of performance. Second, the function has two pieces and the negative piece can be adjusted separately to bring a sufficient penalty for any performance violation. In the other words, parameter α can be set carefully so that if any kind of performance is violated then $u(r,h,d) < 0$ holds. This is stated in the following theorem.

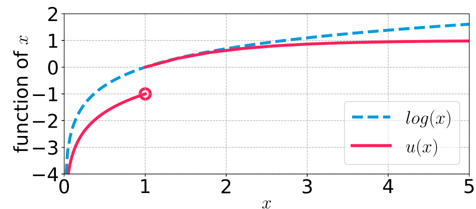


Fig. 3. The utility function $u(x)$ (with α as 1) versus the natural logarithm.

It is noted that we can set the same α for those three functions $u(\frac{r}{r_{\min}})$, $u(\frac{h_{\max}}{h})$, and $u(\frac{d_{\max}}{d})$, but this setting is not tight. Instead, we set α separately and denote them by α_r, α_h and α_d respectively.

Theorem 1. *For all r, h and d, $u(r,h,d) \geq 0$ only if all three kinds of performance requirements are satisfied, if and only if the penalty parameter α in the utility function (6) satisfies the following constraint:*

$$\alpha_r \geq \sigma(\theta_r), \alpha_h \geq \sigma(\theta_h), \text{ and } \alpha_d \geq \sigma(\theta_d), \tag{7}$$

where

$$\sigma(\theta) = \begin{cases} \frac{1-\theta}{\theta}, & \theta \neq 0, \\ 0, & \theta = 0. \end{cases} \tag{8}$$

This can be expressed using logical connectives as follows:

$$\forall r \forall h \forall d \Big(u(r,h,d) \geq 0 \rightarrow \Big(\tfrac{r}{r_{\min}} \geq 1 \wedge \tfrac{h_{\max}}{h} \geq 1 \wedge \tfrac{d_{\max}}{d} \geq 1\Big)\Big)$$
$$\leftrightarrow (\alpha_r \geq \sigma(\theta_r) \wedge \alpha_h \geq \sigma(\theta_h) \wedge \alpha_d \geq \sigma(\theta_d)). \tag{9}$$

Proof. The theorem is logically equivalent to

$$\forall r \forall h \forall d \Big(\Big(\tfrac{r}{r_{\min}} < 1 \vee \tfrac{h_{\max}}{h} < 1 \vee \tfrac{d_{\max}}{d} < 1\Big) \rightarrow u(r,h,d) < 0\Big)$$
$$\leftrightarrow (\alpha_r \geq \sigma(\theta_r) \wedge \alpha_h \geq \sigma(\theta_h) \wedge \alpha_d \geq \sigma(\theta_d)). \tag{10}$$

We prove the forward implication first.

The proof idea is to suppose a combination (r,h,d) makes the disjunction $\frac{r}{r_{\min}} < 1 \vee \frac{h_{\max}}{h} < 1 \vee \frac{d_{\max}}{d} < 1$ true, then $u(r,h,d) < 0$, which implies that 0 is an upper bound of $u(r,h,d)$. We find the supremum of $u(r,h,d)$, then we have the supremum must not be greater than 0 and get the conclusion. There are $2^3 - 1 = 7$ kinds of assignment to (r,h,d) which makes the disjunction $\frac{r}{r_{\min}} < 1 \vee \frac{h_{\max}}{h} < 1 \vee \frac{d_{\max}}{d} < 1$ true. Each corresponds to a supremum.

Suppose $\theta_r \neq 0$ and the resolution requirement is violated, that is $\frac{r}{r_{\min}} < 1$, but the other two performance requirements are met, that is $\frac{h_{\max}}{h} \geq 1$ and $\frac{d_{\max}}{d} \geq 1$. At this time, the supremum of $u(\frac{r}{r_{\min}})$ is $-\alpha_r$, but the supremum of both $u(\frac{h_{\max}}{h})$ and $u(\frac{d_{\max}}{d})$ is 1. Then, we have the supremum of the weighted sum $u(r,h,d)$ is $-\theta_r\alpha_r + \theta_h + \theta_d$. As mentioned above, at this situation 0 is an upper bound of $u(r,h,d)$. Thus, the supremum must satisfy

Authorized licensed use limited to: SHENZHEN UNIVERSITY. Downloaded on August 09,2022 at 09:18:04 UTC from IEEE Xplore. Restrictions apply.

$$-\theta_r\alpha_r + \theta_h + \theta_d \leq 0, \tag{11}$$

and we get

$$\alpha_r \geq \frac{1-\theta_r}{\theta_r} \text{ when } \theta_r \neq 0. \tag{12}$$

Similarly, suppose only frame interval requirement is violated, we get

$$\alpha_h \geq \frac{1-\theta_h}{\theta_h} \text{ when } \theta_h \neq 0. \tag{13}$$

Moreover, suppose only latency requirement is violated, we get

$$\alpha_d \geq \frac{1-\theta_d}{\theta_d} \text{ when } \theta_d \neq 0. \tag{14}$$

Note that the other four kinds of assignment such that the disjunction $\frac{r}{r_{\min}} < 1 \vee \frac{h_{\max}}{h} < 1 \vee \frac{d_{\max}}{d} < 1$ are not needed to analyze because the resulted supremum values are lower than the above three. Putting them together, we have proved the forward implication in (10). The converse can also be easily proved. □

For example, when $\theta_r = \theta_h = 1/2$ and $\theta_d = 0$, we set $\alpha_r = \alpha_h = 1$ according to Theorem 1. As illustrated in Fig. 2b, once $\frac{h_{\max}}{h} < 1$, the value of $\frac{1}{2}u(\frac{r}{r_{\min}}) + \frac{1}{2}u(\frac{h_{\max}}{h})$ becomes negative.

5.3 Penalty of Frame Interval's Variation

As mentioned before, the variation in the frame interval also has a significant impact on user experience. For a sequence of received frames, the less volatile the frame interval is, the better the user experience. A sudden increase in frame interval may interrupt user interactions. We use the relative standard deviation (RSD), which is the ratio of standard deviation to mean, to formulate the variation.

For task s_i, suppose a sequence of m_i frames are received by the user side, let h_{ij} denote the instant frame interval after receiving the jth frame. Then the mean of frame interval for task s_i is

$$E(h)_i = \frac{1}{m_i}\sum_{j=1}^{m_i} h_{ij}, \tag{15}$$

and the mean of frame interval square is

$$E(h^2)_i = \frac{1}{m_i}\sum_{j=1}^{m_i} h_{ij}^2. \tag{16}$$

Let v_i denote the relative standard deviation (RSD) of frame interval for task s_i, then we have

$$v_i = \frac{Std(h)_i}{E(h)_i} = \frac{\sqrt{E(h^2)_i - E(h)_i^2}}{E(h)_i}. \tag{17}$$

With the penalty included, the performance becomes (3).

6 Scheduling Algorithm

We propose an algorithm to address the scheduling problem. As illustrated in Fig. 4, it is composed of a resolution adjustment (RA) algorithm and a frame rate fair (FRF) scheduling algorithm. The former selects resolutions for tasks, and the latter decides the order of tasks to schedule, which is called scheduling sequence. Once a server is available, the processor executes rendering commands in the order of scheduling sequence, with the resolution arrangement determined by the RA algorithm. After many rounds of command executions, the utility is evaluated and fed back to the RA algorithm, which decides how to update the resolution arrangement. The resolution arrangement also affects the FRF scheduling algorithm since one of the algorithm's inputs is the average processing time per command for each task, which is proportional to resolution. The FRF scheduling algorithm further consists of two sub-algorithms: weighted max-min frame rate algorithm and scheduling sequence algorithm. They are called in sequence, where the former gets a scheduling frequency vector f (the number of command executions per time unit for each task) and then the latter gets a scheduling sequence vector S.

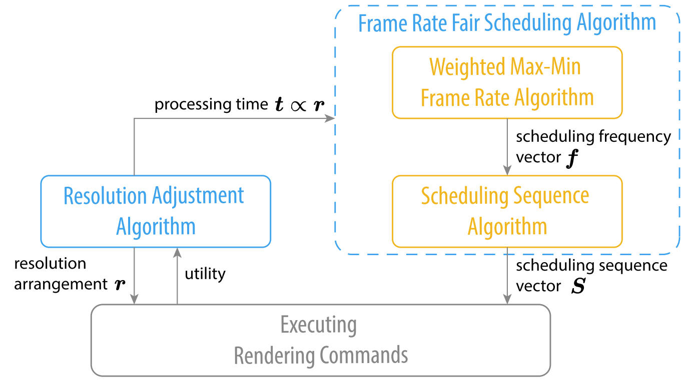


Fig. 4. The framework of our algorithm.

6.1 Resolution Adjustment Algorithm

For each task, a set of k resolutions could be used. A low resolution may not meet the resolution requirement, while a high resolution may lead to long processing time and transmission time, further results in long latency and may violate the latency constraint. Thus, when there is a single task, its utility is a concave function of resolution. However, when multiple tasks contend against each other, how an individual resolution decision affects the final utility (which is the minimum over all tasks) is complicated.

Given n tasks and k resolutions, each task chooses a resolution, and we can get n^k permutations, each representing a resolution arrangement. A naive way is to try all of them and select the best one. However, it takes $O(n^k)$ rounds. Moreover, trying out too many bad arrangements degrade overall performance. To address these issues, we start from an initial arrangement and gradually elevate it until the overall utility does not improve anymore, so as to avoid bad arrangements as much as possible.

For such an algorithm, three design issues matter. The first is the initial resolution; the second is how to elevate a resolution arrangement; the third is the convergence condition. We discuss each of them in the following. The initial resolution for each task should be the resolution requested by the user, using which can meet the resolution requirement. When computation resource is abundant, promoting

Authorized licensed use limited to: SHENZHEN UNIVERSITY. Downloaded on August 09,2022 at 09:18:04 UTC from IEEE Xplore. Restrictions apply.

resolution for some tasks may improve the overall performance. Among all tasks, the task with the lowest utility should be favored as it limits the overall performance. Thus, we elevate a resolution arrangement by promoting the resolution of the worst-performing task by one level. We keep raising resolutions until the overall utility drops due to insufficient computation resource, when the algorithm is deemed to be converged. However, the randomness in the running system may result in performance fluctuation, which may mislead the algorithm into making premature convergence. To tolerate this uncertainty, we only allow a significant utility reduction to trigger convergence. Note that the trial should continue when the utility has no change, since ties may exist and in that situation promoting a single resolution may not improve the minimum performance but promoting all resolutions can.

Another critical question is when to adjust resolution. To evaluate the performance of a resolution arrangement as accurately as possible, we have to maintain each resolution arrangement long enough before changing it. Besides, video encoding complicates the question a bit. The rendered frames are usually encoded in groups of pictures (GOP). Each GOP includes an intra-coded frame, which is encoded independently, and many inter-coded frames, which are encoded depending on the previous frames in the same GOP. For a group of pictures, the same resolution should be used. Otherwise, normal video encoding techniques (e.g., H.264, MPEG-4, *etc.*) cannot support. Thus, for each task, only when a GOP finishes, can its resolution be adjusted. Taking the above two aspects into consideration, a resolution adjustment is initiated when two conditions are met simultaneously: 1) enough time has passed since the last adjustment; 2) a GOP has finished encoding for the task to adjust. In our simulation later, the GOP length is 64 frames, and the resolution maintaining time is set to the time for rendering 1024 frames, which is counted over all tasks.

We present the resolution adjustment algorithm in Algorithm 1. Let $\boldsymbol{r}$ denote the resolution arrangement set currently, which is a vector, each element of whom is a resolution for a task. Our algorithm starts from the initial arrangement $\boldsymbol{r}_{\min}$ (the required resolutions) and runs iteratively. As mentioned above, each iteration is initiated by two conditions. In each iteration, we first get the task which initiates this iteration, say s_k (line 5), since only this task has an opportunity to adjust its resolution in this round. Next, we compute the average utility $\tilde{u}_i$ for every task since the last resolution adjustment (i.e., the last change of $\boldsymbol{r}$) according to (3), as well as the overall utility $\tilde{u}$ (line 6). Actually, $\tilde{u}$ is a performance measurement of the current resolution arrangement $\boldsymbol{r}$. If the algorithm has been deemed to be converged, then we recover the resolution of the initiation task according to the best solution found so far (denoted by $\boldsymbol{r}^*$) (lines 7-8). Otherwise, if the initiation task s_k gets the least utility, then we decide whether to elevate $\boldsymbol{r}$ or terminate (lines 9-23). It is noted that if the initiation task does not lead to the least utility, then we are not about to promote its resolution, since the promotion is unable to improve the least utility.

Specifically, the algorithm in lines 9-23 proceeds in three steps. First, we record the best solution found so far in order to recover it when necessary (lines 10-14), where $\tilde{u}^*$ and $\boldsymbol{r}^*$ are the maximum utility and the best resolution arrangement found so far respectively. Second, we elevate $\boldsymbol{r}$ by promoting the initiation task s_k's resolution by one level, if the following three conditions are satisfied simultaneously: 1) $\tilde{u}$ does not drop significantly relative to $\tilde{u}^*$ (i.e., $\tilde{u} \geq \tilde{u}^* - \xi$, where $\xi \geq 0$); 2) with the current resolution arrangement $\boldsymbol{r}$, all the constraints (4b)-(4d) are met; 3) the task s_k's resolution still has room to increase (lines 15-18). Here, the parameter ξ serves as a margin, which prevents the algorithm from converging prematurely due to fluctuations in utility. It is noticed that in evaluating the constraints (4b)-(4d) we compute the average performance for the commands processed with the current resolution configuration $\boldsymbol{r}$. Finally, the algorithm converges otherwise. At this situation, we recover the best solution found before (lines 19-22).

Algorithm 1. Resolution Adjustment Algorithm

```
1: Initialize converged ← False
2: Initialize r ← r_min
3: Initialize r* ← null, ũ* ← −∞
4: while resolution adjustment is initiated by a task do
5:   Get the initiation task s_k
6:   Get {ũ_i, ∀i} and ũ ← min_{i=1,...,n} ũ_i
7:   if converged = True then
8:     Recover the resolution of s_k according to r*
9:   else if k = arg min_{i=1,...,n} ũ_i then
10:    Record the best solution found so far:
11:    if ũ > ũ* then
12:      ũ* ← ũ
13:      r* ← r
14:    end if
15:    if ũ ≥ ũ* − ξ and
16: all constraints in (4b)-(4d) are met and
17: s_k's resolution is not the maximum then
18:      Elevate r by promoting task s_k's resolution
19:    else
20:      converged ← True
21:      Recover the resolution of s_k according to r*
22:    end if
23:  end if
24: end while
```

The rendering tasks served by an edge server change dynamically, so the computing load will also change accordingly. Thus, when a new task starts or an old task finishes, we restart the resolution adjustment algorithm so that the resolution arrangement can adapt to changes.

In the resolution adjustment algorithm, in each iteration, we compute the utility $\tilde{u}_i$ for every task according to (3), and calculate the least utility (i.e., $\min_{i=1,\ldots,n}\tilde{u}_i$). The constraints (4b)-(4d) are also evaluated. Thus, the time complexity at each iteration is $O(n)$.

6.2 Frame Rate Fair Scheduling Algorithm

In this section, we propose a scheduling algorithm to optimize QoS.

6.2.1 *Scheduling Algorithm*

Given a set of n tasks $\{s_1, s_2 \ldots s_n\}$, we may schedule them in a round robin way, and then each task would get similar rendering frequencies and frame rates. However, this is

Authorized licensed use limited to: SHENZHEN UNIVERSITY. Downloaded on August 09,2022 at 09:18:04 UTC from IEEE Xplore. Restrictions apply.

unable to achieve the best fairness when tasks have distinctive requirements for frame rate/interval, and may also violate the frame interval constraint for some tasks with high requirements. Instead, we optimize scheduling frequency for each task to achieve weighted max-min fairness [33], [34] in terms of frame rate. We call this *weighted max-min frame rate problem*. A frame rate allocation is max-min fair if one cannot increase the frame rate of a task without decreasing that of another task which already has a lower frame rate.

Our goal is to achieve the fairness of the long-term frame rate, but the scheduling frequency of a task is only planned for the next period, which decides short-term frame rate. The relationship between the short-term scheduling frequency and the long-term frame rate is derived as follows. For task s_i, let f_i denote the expected scheduling frequency in the coming period, whose duration will be decided later. Suppose the average frame rate achieved so far, which is the long-term frame rate, is $\bar{f}_i$, then after the next period the achieved long-term frame rate, denoted by x_i, becomes

$$x_i = (1 - \beta) f_i + \beta \bar{f}_i, \tag{18}$$

where β is a constant parameter. Then, the long-term frame rate ratio is $x_i / f^i_{\min}$. *The weighted max-min frame rate problem is to find the weighted max-min fair vector* $\boldsymbol{x}$ *on a feasible set*, which will be analyzed later. According to [33], [34], weighted max-min fairness is defined as follows. Given some positive weights $f^i_{\min}$, a vector $\boldsymbol{x}$ is weighted max-min fair on a feasible set, if and only if increasing one component x_s must be at the expense of decreasing some other component x_t such that $x_t / f^t_{\min} \leq x_s / f^s_{\min}$.

With the weighted max-min fair $\boldsymbol{x}$, we can get short-term frame rate vector $\boldsymbol{f}$ from (18). Next, we determine the duration of the next period. In order to make sure that every task with nonzero f_i is scheduled at least once, we set the period duration to $1/\min_{i|f_i>0} f_i$. As a result, the number of scheduling times for task s_i in the next period, denoted by Q_i, is

$$Q_i = \frac{f_i}{\min_{i|f_i>0} f_i}. \tag{19}$$

Once the number of scheduling times for each task is known, we may schedule tasks in a round robin way according to these quanta. However, it may perform poorly because it may lead to large fluctuations of frame interval. In order to smooth out the changes in frame intervals, we have to optimize the order of tasks to schedule, which is called scheduling sequence. We formulate it to be a vector, whose jth element is task s_k if s_k will be processed in jth step. For example, given three tasks s_1, s_2 and s_3, suppose under a plan the three tasks have 1, 2 and 3 commands to execute respectively, then vector $(s_1, s_2, s_3, s_2, s_3, s_3)$ and vector $(s_1, s_3, s_2, s_3, s_2, s_3)$ are two types of scheduling sequence. Suppose a command's processing time is the same for every task, then for the task s_3, in the former case there are two frame interval samples: 1 and 0; in comparison, in the latter case there are two frame interval samples: 1 and 1. The latter is better than the former in terms of frame interval's variation. We define *the scheduling sequence problem: given a set of commands executed in a period, to optimize the scheduling sequence of these commands such that the penalty of frame interval's variation is minimized.*

The weighted max-min frame rate problem and the scheduling sequence problem are two core issues in scheduling. With each of them solved, our scheduling algorithm is easy to design. As shown in Algorithm 2, our scheduling algorithm proceeds when the processor is available and at least one task has commands in the queue. It proceeds in a periodical way. At the beginning of a new period, the algorithm proceeds in four steps:

1) solve the weighted max-min frame rate problem and obtains a scheduling frequency vector $\boldsymbol{f}$ (line 6);
2) get the amount of planned command execution (denoted by m_i for task s_i) from $\boldsymbol{f}$ (lines 7-10);
3) solve the scheduling sequence problem and obtains a scheduling sequence vector $\boldsymbol{S}$ with $\boldsymbol{m}$ (a vector whose ith element is m_i) as input (line 11);
4) reset the iterator $next$ of $\boldsymbol{S}$ to zero.

In the second step above, we get m_i by accumulating Q_i in a deficit value D_i and rounding down D_i, instead of directly rounding down Q_i. The reason is that with the latter for some tasks the actual achieved frame rate would be always lower than the planned value f, resulting in poor performance. We will demonstrate this in the section of performance evaluation.

During a period, our algorithm traverses the vector $\boldsymbol{S}$ to find the next task which has commands and whose deficit value is not less than one (lines 14-17). Then, it schedules the chosen task s_{idx} and reduces the associated deficit value by one (lines 18-19). Once a traversal over the vector $\boldsymbol{S}$ finishes, a new period starts (line 5).

Algorithm 2. Frame Rate Fair Scheduling Algorithm

1: Initialize $D_i \leftarrow 0, \forall i$
2: Initialize $next \leftarrow 1$
3: Initialize $m \leftarrow 0$
4: **while** the processor is available **and** at least one task has a command **do**
5: **if** $next > m$ **then** ▷ start a new period
6: $\boldsymbol{f} \leftarrow$ GETSCHEDULINGFREQ($\bar{\boldsymbol{f}}, \boldsymbol{f}_{\min}, \boldsymbol{\lambda}, \boldsymbol{t}, \beta$)
7: $Q_i \leftarrow f_i / \min_{i|f_i>0} f_i, \forall i$
8: $D_i \leftarrow D_i + Q_i, \forall i$
9: $m_i \leftarrow \lfloor D_i \rfloor, \forall i$
10: $m \leftarrow \sum_i m_i$
11: $\boldsymbol{S} \leftarrow$ GETSCHEDULINGORDER($\boldsymbol{m}, \boldsymbol{t}, \boldsymbol{\tau}_0$)
12: $next \leftarrow 0$
13: **end if**
14: **repeat**
15: $next \leftarrow next + 1$
16: $s_{idx} \leftarrow \boldsymbol{S}[next]$
17: **until** task s_{idx} has commands **and** $D_{idx} \geq 1$
18: Schedule task s_{idx}
19: $D_{idx} \leftarrow D_{idx} - 1$
20: **end while**

6.2.2 Weighted Max-Min Frame Rate Problem & Algorithm

In this section, we discuss how to formulate and solve the weighted max-min frame rate problem. We first formulate

Authorized licensed use limited to: SHENZHEN UNIVERSITY. Downloaded on August 09,2022 at 09:18:04 UTC from IEEE Xplore. Restrictions apply.

the feasible set of $\boldsymbol{x}$. The short-term frame rate f_i is nonnegative and must not be greater than the arrival rate of commands, denoted by λ_i, that is $0 \leq f_i \leq \lambda_i$. With (18), we have

$$\beta \bar{f}_i \leq x_i \leq (1-\beta)\lambda_i + \beta \bar{f}_i, \quad \forall i = 1 \dots n. \tag{20}$$

The arrival rate of commands can be estimated via statistical analysis. The time consumed by running all commands is $\sum_i t_i f_i$, where t_i is the average processing time per command for task s_i. Let the period be one second, then we have

$$\sum_{i=1\dots n} t_i f_i \leq 1. \tag{21}$$

With (18), we have

$$\sum_{i=1\dots n} t_i x_i \leq 1 - \beta + \beta \sum_{i=1\dots n} t_i \bar{f}_i. \tag{22}$$

Thus, the feasible set of $\boldsymbol{x}$ is

$$\left\{ \boldsymbol{x} \,\middle|\, \sum_{i=1\dots n} t_i x_i \leq 1 - \beta + \beta \sum_{i=1\dots n} t_i \bar{f}_i, \right. \\ \left. \beta \bar{f}_i \leq x_i \leq (1-\beta)\lambda_i + \beta \bar{f}_i, \ \forall i = 1 \dots n \right\}. \tag{23}$$

It is convex and compact. Our objective is to achieve the weighted max-min fair allocation $\boldsymbol{x}$ on the set (23), where x_i's weight is $f^i_{\min}$. According to the theory on max-min fairness, it exists and can be found by the water filling algorithm [33], [34].

Algorithm 3. Optimizing Scheduling Frequency

1: **procedure** GetSchedulingFreq ($\bar{\boldsymbol{f}}, \boldsymbol{f_{\min}}, \boldsymbol{\lambda}, \boldsymbol{t}, \beta$)
2: Get parameters C, L_i, U_i, W_i
3: $y_i \leftarrow L_i, \ \forall i = 1 \dots n$
4: $C' \leftarrow C - \sum_{i=1}^n y_i$
5: $\mathbf{I} \leftarrow \{1, 2, \dots, n\}$
6: **while** $C' > 0$ **and** $\mathbf{I} \neq \emptyset$ **do**
7: Get the tasks $\mathbf{I}'$ with the smallest y_i/W_i from $\mathbf{I}$
8: $V_1 \leftarrow$ the smallest y_i/W_i among $\mathbf{I}$
9: $V \leftarrow V_1 + \frac{C'}{\sum_{i \in \mathbf{I}'} W_i}$
10: **if** $\mathbf{I}' \neq \mathbf{I}$ **then**
11: $V_2 \leftarrow$ the second smallest y_i/W_i among $\mathbf{I}$
12: $V \leftarrow \min\{V, V_2\}$
13: **end if**
14: $y_i \leftarrow \min\{VW_i, U_i\}, \quad \forall i \in \mathbf{I}'$
15: $C' \leftarrow C - \sum_{i=1}^n y_i$
16: $\mathbf{I} \leftarrow \{i | y_i < U_i\}$
17: **end while**
18: $x_i \leftarrow y_i/t_i, \forall i = 1 \dots n$
19: $f_i \leftarrow (x_i - \beta \bar{f}_i)/(1-\beta), \forall i = 1 \dots n$
20: Return scheduling frequency vector $\boldsymbol{f}$
21: **end procedure**

In order to present the algorithm more clearly, we transform the above problem into a concise form. Let $y_i = t_i x_i$, then the problem is equivalent to find the weighted max-min fair allocation $\boldsymbol{y}$ in the feasible set

$$\left\{ \boldsymbol{y} \,\middle|\, \sum_{i=1\dots n} y_i \leq C, \quad L_i \leq y_i \leq U_i, \quad \forall i = 1 \dots n \right\}, \tag{24}$$

where capacity $C = 1 - \beta + \beta \sum_{i=1\dots n} t_i \bar{f}_i$, lower bound $L_i = \beta t_i \bar{f}_i$, and upper bound $U_i = (1-\beta) t_i \lambda_i + \beta t_i \bar{f}_i$. The weight of y_i is $W_i = t_i f^i_{\min}$.

We illustrate the water filling algorithm in Algorithm 3. Its inputs include $\bar{\boldsymbol{f}}$, $\boldsymbol{f_{\min}}$, $\boldsymbol{\lambda}$ and $\boldsymbol{t}$, each of which is a vector composed of $\bar{f}_i$, $f^i_{\min}$, λ_i and t_i respectively. The algorithm first gets the parameters as shown above (line 2). The algorithm starts the rate (i.e., y_i) of each task with the associated lower bound, and then iteratively increases each rate until either it reaches the associated upper bound or total capacity runs out (lines 3-17). Let C' denote the remaining capacity, which is initialized to $C - \sum_{i=1}^n L_i$, and $\mathbf{I}$ denote the set of tasks whose rate can be increased further, which is initialized to $\{1, 2, \dots, n\}$. In each iteration, the algorithm first finds the subset of tasks with the smallest weighted rate (i.e., y_i/W_i) among $\mathbf{I}$, which is denoted by $\mathbf{I}' \subseteq \mathbf{I}$ (line 7). Second, it increases the rates of the tasks in $\mathbf{I}'$ in a fair way, which maintains the tasks' weighted rates the same until a certain level (lines 8-14). Third, it updates the remaining capacity C' and removes the tasks that reach their upper bounds from $\mathbf{I}$ (lines 15-16).

Next, we discuss how to increase the rates of the tasks in $\mathbf{I}'$ (lines 8-14) in detail. The algorithm keeps increasing until one of the following events takes place: 1) the remaining capacity runs out; 2) the weighted rate reaches the second smallest value among $\mathbf{I}$, since at that time the subset $\mathbf{I}'$ changes. In the former case, the weighted rate can reach

$$V_1 + \frac{C'}{\sum_{i \in \mathbf{I}'} W_i}, \tag{25}$$

where V_1 denotes the smallest weighted rate among $\mathbf{I}$. In the latter case, the weighted rate can reach the second smallest weighted rate among $\mathbf{I}$, denoted by V_2. It is noticed that V_2 does not exist when $\mathbf{I}' = \mathbf{I}$. Overall, the target weighted rate (denoted by V) to reach is the minimum one between them (lines 8-13). Then, as shown in line 14, the rate of each task in $\mathbf{I}'$ should become

$$y_i = \min\{VW_i, U_i\}, \quad \forall i \in \mathbf{I}'. \tag{26}$$

After obtaining y_i by the water-filling algorithm, we get $x_i = y_i/t_i$ and f_i according to (18).

6.2.3 Scheduling Sequence Problem & Algorithm

In this section, we introduce how to formulate and address the scheduling sequence problem. Given m commands executed in a period, among which m_i commands are for task s_i, the duration of a period is $T = \sum_i m_i t_i$, where t_i is the average processing time per command for task s_i. The ordering problem is to arrange m commands within the period $[0, T]$ such that the penalty of frame interval's variation is minimized. The objective is

$$\text{minimize} \quad \max_{i=1,\dots,n} \quad v_i, \tag{27}$$

where v_i is the relative standard deviation of frame interval for task s_i as defined in (17).

It is noticed that in the formulation introduced in Section 5, frame interval is measured at the user side. However, in the scheduling sequence problem, we plan command

Authorized licensed use limited to: SHENZHEN UNIVERSITY. Downloaded on August 09,2022 at 09:18:04 UTC from IEEE Xplore. Restrictions apply.

execution rather than implement it, thus we cannot obtain frame interval in advance. Instead, we use the time interval between the planned starts of two commands to approximate it.

The problem is a combinatorial optimization. We propose a heuristic algorithm to solve it. The idea is to distribute m_i commands within a period as evenly as possible so as to smooth out the changes in frame interval. Specifically, for task s_i, we try to arrange starting time for m_i commands such that the gap between two consecutive starts is close to

$$g_i = \frac{T}{m_i}. \tag{28}$$

Let o_{ij} denote the jth command of task s_i and τ_{ij} denote its starting time. Then, we use $\tau_{ij} - \tau_{i,j-1}$ to approximate frame interval h_{ij}. For command o_{ij}, with its starting time τ_{ij} arranged, its finishing time can be approximated as $\tau_{ij} + t_i$. As a result, $[\tau_{ij}, \tau_{ij} + t_i]$ becomes a busy interval and has to be removed from the period $[0, T]$. It is noted that the intervals mentioned in this section are left-closed and right-open. As commands are placed, the period becomes an ordered set of idle intervals separated by busy intervals having been arranged. Let $\mathbf{E} = \{[e_k^s, e_k^f], k = 1, 2 \ldots\}$ denote it, where e_k^s and e_k^f are the starting time and the finishing time of the kth idle interval respectively.

Given an idle interval set $\mathbf{E}$, we determine starting time τ_{ij} for a command o_{ij}. First, we ensure that since the last command $o_{i,j-1}$ was launched at least g_i time has passed, that is

$$\tau_{ij} \geq \tau_{i,j-1} + g_i. \tag{29}$$

Next, we try to place the command in idle intervals rather than busy ones, so as to mitigate the changes on those commands having been arranged. There are two situations where the command o_{ij} can be inserted into an idle interval $[e_k^s, e_k^f]$: 1) when $\tau_{i,j-1} + g_i$ is within the idle interval, we can arrange the command here and set τ_{ij} to $\tau_{i,j-1} + g_i$; 2) when the idle interval starts later than $\tau_{i,j-1} + g_i$, we can also arrange the command and set τ_{ij} to e_k^s. Thus, we have

$$\tau_{ij} = \begin{cases} \tau_{i,j-1} + g_i, & e_k^s \leq \tau_{i,j-1} + g_i < e_k^f, \\ e_k^s, & e_k^s > \tau_{i,j-1} + g_i. \end{cases} \tag{30}$$

There may be several idle intervals available for placing a command. The number of choices is $O(m)$. An interesting question is which one should we choose. Different choices may lead to different objective values (27). Usually, the earliest one leads to the highest objective value, because it results in the smallest variation in frame interval for the current task. Thus, we directly take the earliest one. Alternatively, we may try each choice and choose the one leading to the highest objective value. However, it is time-consuming. We compare these two options via simulations and find their performance are very close.

A new command arrangement may influence existing ones. Suppose a command o_{ij} is inserted into an idle interval $[e_k^s, e_k^f]$, and its busy interval is $[\tau_{ij}, \tau_{ij} + t_i]$. If the busy interval is beyond the idle one, that is $\tau_{ij} + t_i > e_k^f$, then the new busy interval must overlap with other existing ones,

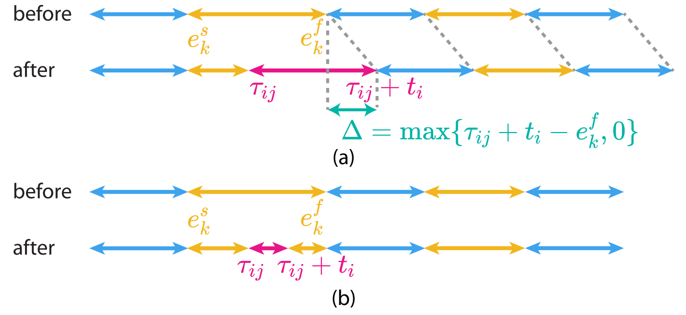


Fig. 5. A set of busy intervals are shown in blue, and a set of idle intervals are shown in yellow. A command o_{ij}, whose busy interval is $[\tau_{ij}, \tau_{ij} + t_i]$ (in magenta), is inserted into an idle interval $[e_k^s, e_k^f]$. (a) if the busy interval is beyond the idle one, we need to shift the intervals (both busy and idle) starting later than e_k^f by Δ, and the idle interval is truncated. (b) otherwise, shifting is unnecessary, but the idle interval is divided into two.

and we have to adjust some existing arrangements to avoid such overlapping. Specifically, as illustrated in Fig. 5a, we shift the intervals (both busy and idle) starting later than e_k^f by

$$\Delta = \max\{\tau_{ij} + t_i - e_k^f, 0\}. \tag{31}$$

Thus, we have

$$\tau_{i'j'} \mathrel{+}= \Delta, \text{ if } \tau_{i'j'} \geq e_k^f, \quad \forall i' \forall j'. \tag{32}$$

Similarly, we shift idle intervals $[e_{k'}^s, e_{k'}^f]$ as follows:

$$[e_{k'}^s, e_{k'}^f] = [e_{k'}^s + \Delta, e_{k'}^f + \Delta], \text{ if } e_{k'}^s > e_k^f, \quad \forall k'. \tag{33}$$

Once a command is arranged, the idle interval set $\mathbf{E}$ has to be updated accordingly. Except the above shifting, the idle interval $[e_k^s, e_k^f]$ becomes as follows:

$$[e_k^s, e_k^f] = \begin{cases} [e_k^s, \tau_{ij}] \text{ and } [\tau_{ij} + t_i, e_k^f], & \tau_{ij} + t_i \leq e_k^f, \\ [e_k^s, \tau_{ij}], & \text{o.w.} \end{cases} \tag{34}$$

That is, as illustrated in Fig. 5b, if the new busy interval $[\tau_{ij}, \tau_{ij} + t_i]$ is totally within the idle interval, that is $\tau_{ij} + t_i \leq e_k^f$, then the idle interval is divided into two segments $[e_k^s, \tau_{ij}]$ and $[\tau_{ij} + t_i, e_k^f]$; otherwise, as illustrated in Fig. 5a, the original idle interval is truncated and becomes $[e_k^s, \tau_{ij}]$.

Algorithm 4. Optimizing Scheduling Sequence

1: **procedure** GETSCHEDULINGORDER $(\boldsymbol{m}, \boldsymbol{t}, \boldsymbol{\tau}_0)$
2: $\quad T \leftarrow \sum_i m_i t_i$
3: $\quad \mathbf{E} \leftarrow \{[0, T]\}$
4: $\quad g_i \leftarrow T/m_i, \forall i$
5: $\quad$ Sort $\{s_i\}$ in the decreasing order of m_i
6: $\quad$ **for** s_i in the sorted order **do**
7: $\quad\quad$ **for** $j \leftarrow 1, m_i$ **do**
8: $\quad\quad\quad$ Get the first available idle interval, say $[e_k^s, e_k^f]$
9: $\quad\quad\quad$ Compute τ_{ij} via (30)
10: $\quad\quad\quad$ Update $\boldsymbol{\tau}$ via (32)
11: $\quad\quad\quad$ Update $\mathbf{E}$ via (33) and (34)
12: $\quad\quad$ **end for**
13: $\quad$ **end for**
14: $\quad$ Get & return scheduling sequence S according to $\boldsymbol{\tau}$
15: **end procedure**

Authorized licensed use limited to: SHENZHEN UNIVERSITY. Downloaded on August 09,2022 at 09:18:04 UTC from IEEE Xplore. Restrictions apply.

We present the scheduling sequence algorithm in Algorithm 4. Its inputs include $\boldsymbol{m}$, $\boldsymbol{t}$ and $\boldsymbol{\tau_0}$, each of which is a vector composed of m_i, t_i and τ_{i0} respectively. τ_{i0} is the starting time (relative to current time) of the latest executed command (the one before the first planned command) for task s_i. It is noted that τ_{i0} is a negative value. With these inputs, we initialize variables (lines 2-4).

Our algorithm deals with n tasks in the decreasing order of command amount. Our rationale is that the more commands a task has, the lower its target average frame interval g_i is, and the more volatile the frame interval is. For each task s_i, we determine starting time for m_i commands in turn. For each command o_{ij}, we take the earliest idle interval available for placing the command from $\mathbf{E}$, say $[e_k^s, e_k^f]$ (line 8). Then, we compute τ_{ij} via (30) and update the starting time of the commands previously arranged (lines 9-10). Let $\boldsymbol{\tau}$ denote starting time matrix composed of τ_{ij}. We update $\boldsymbol{\tau}$ via (32). Fourth, we update the idle interval set $\mathbf{E}$ by shifting according to (33) and removing the new busy interval according to (34) (line 11). Finally, after all commands are arranged, we get the scheduling sequence S according to the latest $\boldsymbol{\tau}$ and return it (line 14).

Note that if t_i are the same for all tasks, then the above algorithm will become simpler. We can consider the whole period as m equal-length time slots, whose length is the average processing time per command, and each command will be allocated to a slot. As such, the complicated situations as shown in Fig. 5 will not appear, thus line 10 in Algorithm 4 is unnecessary and line 11 becomes simpler. Now, the idle interval set $\mathbf{E}$ maintains the set of idle slots, which is initialized to include all time slots. Once a slot is selected for a command, it is simply removed from $\mathbf{E}$.

7 Simulation Setup

To make our simulation more practical, we produce simulation environments using trace data. In following, we present how to collect trace data and how to produce simulation environments.

7.1 Trace Data Collection

We use a game engine called Unity [35] to render an animated 3D scene [36] in different resolutions and trace the processing time. We run it using Nvidia GeForce GTX 1060. The reason why we do not use a higher-performance GPU is that Unity cannot provide sufficient precision to trace very short processing time. Instead, we use a medium performance GPU to collect data and then reduce the processing time by a factor to simulate rendering via high-performance GPU. The factor we used is 0.3.

Ideally, we should render each frame in multiple resolutions and time each rendering operation. However, implementing in this way, we find it is hard to obtain the accurate time for each rendering operation because Unity has a lot of built-in caching and threading mechanisms. Instead, we find empirically that the time required to render a frame is approximately linearly related to the resolution, so we first render the scene in a selected resolution, namely the baseline resolution, and then scale its rendering time to obtain the rendering time of other resolutions. In the following simulation, the resolution candidate set is $\{1920 \times 1080, 2560 \times 1440, 3072 \times 1728, 3840 \times 2160\}$, among which 2560×1440 is chosen as the baseline resolution. The scaling factors for each of the four resolutions are set to 0.73, 1.0, 1.37, and 2.18 respectively. The scaling factors have been obtained empirically, by first rendering the scene in different resolutions independently, namely raw data, and then dividing their median rendering time by that of the baseline resolution. We illustrate the empirical cumulative distribution function (CDF) of the processing time in Fig. 6. The generated data has similar statistical features to the raw data.

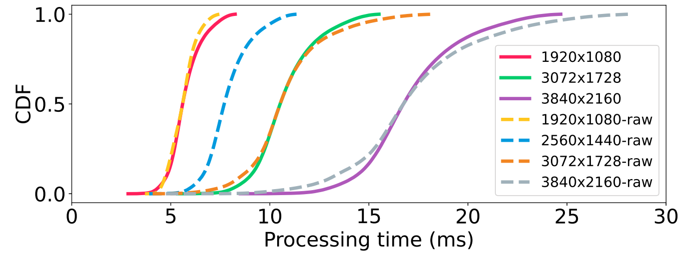


Fig. 6. The empirical CDF of the processing time samples rendered in various resolutions. The generated data is shown in solid line, while the raw data is shown in dash line. For every resolution except the baseline (2560×1440) the generated data has similar statistical features with the raw data.

7.2 Simulation Environment

A simulation environment consists of n tasks, each having a set of QoS requirements and a sequence of rendering commands.

7.2.1 QoS Requirements

As mentioned in Section 4.1, trade-offs should be considered in defining QoS requirements. Here, we present a simple approach. First, we formulate the resolution-latency trade-off as below:

$$d_{\max} \geq t_p(r_{\min}) + t_x(r_{\min}), \tag{35}$$

where $t_p(r)$ and $t_x(r)$ denote the average processing time and transmission time for resolution r respectively. Otherwise, the latency limit $d_{\max}$ cannot be met. We formulate $t_x(r) = \frac{r\,b\,c}{B}$, where b is bits per pixel, c is compression ratio, and B is bandwidth. Second, we formulate the frame rate-latency trade-off as below:

$$f_{\min} \leq \frac{1}{t_p(r_{\min}) + t_x(r_{\min})}. \tag{36}$$

In this conservative estimation, we ignore the parallelism of processing and transmitting for simplicity.

We generate QoS requirements as follows. First, we randomly select a bandwidth value B, $f_{\min}$, $r_{\min}$ from a corresponding candidate set following a uniform distribution. These candidate sets are listed in Table 1. Second, rather than setting $d_{\max}$ (in ms) to a single value, we set

$$d_{\max} = \min\{80, \mathrm{round}\left(\frac{t_p(r_{\min}) + t_x(r_{\min})}{10}\right) \cdot 10 + 30\}. \tag{37}$$

The higher $d_{\max}$ is, the larger the opportunity for promoting resolution. In Fig. 7, we show the statistics of latency limit in our simulation. There are 1349 samples in total, the median is 40 ms and the mean is 46.3 ms. Third, we filter

Authorized licensed use limited to: SHENZHEN UNIVERSITY. Downloaded on August 09,2022 at 09:18:04 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
Parameter Setting for the Simulation Environment

Environment Parameter	Value
Simulation duration (s)	120
Bits per pixel b	32
Encoding time (ms)	$\mathrm{Unif}\{2, 6\}$
GOP length (frames)	64
c for intra-coded frames	$1:\mathrm{Unif}\{400, 600\}$
c for inter-coded frames	$1:\mathrm{Unif}\{800, 1200\}$
The candidate set of $f^i_{\min}$	$\{18, 24, 30, 40, 50, 60\}$ (FPS)
The candidate set of $r^i_{\min}$	$\{1920\times1080, 2560\times1440, 3072\times1728, 3840\times2160\}$
Tolerable latency $d^i_{\max}$ (ms)	$\min\{80, \mathrm{round}(\frac{t_p(r_{\min})+t_x(r_{\min})}{10})\cdot 10+30\}$
The candidate set of mean bandwidth (Mbps)	$\{10, 20, 30\}$
The candidate set of n	$\{2, 3, 4\}$

$\mathrm{Unif}\{a, b\}$ *denotes discrete uniform distribution between a and b.*

the QoS requirements meeting the above conditions (35) and (36). Parameters b and c are set as listed in Table 1. Finally, we set $h_{\max}$ to $1/f_{\min}$ seconds.

7.2.2 Task Assignment

Next, we simulate the task assignment and produce a set of tasks assigned on a server, which is called a task group. We define the computing load of a task as the portion of time spent on processing per second. For a task, given $f_{\min}$ and $r_{\min}$, let L denote the computing load, and then we have

$$L = f_{\min} \cdot t_p(r_{\min}), \tag{38}$$

The load is actually the minimum computing power required for preset QoS. Suppose n tasks are assigned on a server, each having bandwidth B_i, QoS requirements $(f^i_{\min}, h^i_{\max}, r^i_{\min}, d^i_{\max})$ and load L_i, then we have two constraints for bandwidth and load

$$\sum_{i=1}^{n} B_i \leq B_{\max}, \quad \text{and} \quad \sum_{i=1}^{n} L_i \leq L_{\max}, \tag{39}$$

where $B_{\max}$ and $L_{\max}$ are bandwidth limit (100 Mbps in the simulation) and load limit (1 in the simulation) respectively. Other resources can be formulated in the same manner. For simplicity, we assume a homogeneous setting, and thus other resource constraints can be transformed to the limitation on the number of tasks. We limit the number of tasks between 2 and 4 since it is trivial to schedule a single task and the total load of 5 tasks is far beyond the load limit.

We generate task groups as follows. First, we generate possible QoS requirements (together with bandwidth) as introduced above. Second, we generate the combinations of those QoS requirements and filter those satisfying (39). Here, each combination corresponds to a task group. Third, from these task groups, we choose some to simulate based on their total loads. Specifically, given a set of loads, i.e., $\{30\%, \ldots, 100\%\}$, for each level, we randomly select the task groups whose loads are close to it (allowing for $\pm1.5\%$ fluctuation). Here 30% is the lowest possible load.

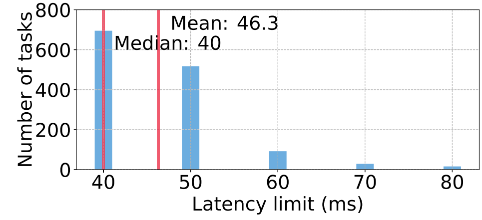


Fig. 7. The histogram of latency limit.

7.2.3 Producing Commands

For a task, we produce a sequence of commands whose arrival rate is higher than its required frame rate. Given the required frame rate $f_{\min}$, the arrival intervals are uniformly distributed between $\frac{0.8}{3f_{\min}}$ seconds and $\frac{1.2}{3f_{\min}}$ seconds. We simulate the arrival of commands lasting 120 seconds. The processing time of each command is sequentially read from a stream of trace data, which is the concatenation of multiple randomly picked segments (5000 samples per segment) of the original trace data, as generated by the method mentioned in Section 7.1. This random composition of trace data segments allows for more variations to be tested.

In addition, we simulate dynamic bandwidth. We collect three sets of bandwidth trace data using Continuous Speed Test Tool [37], each for one of the bandwidth candidates (10, 20, and 30 Mbps). For each set of data, we first divide it into 50 segments, each consisting of bandwidth samples lasting 120 seconds, and then shift every segment of data such that its mean value equals the designated bandwidth (10, 20, and 30 Mbps). The instant bandwidth for each frame to transmit is sequentially read from a randomly picked segment.

Besides, we simplify video encoding to a short process. The encoding time per frame is randomly chosen according to a uniform distribution between 2 and 6 ms. The GOP length is 64 frames. The compression ratio is 1:x, where x is an integer value uniformly distributed in [400, 600] for intra-coded frames and in [800, 1200] for inter-coded frames respectively. The other parameters for the simulation environment are set as listed in Table 1.

8 Performance Evaluation

We do simulations to evaluate our method: the Frame Rate Fair scheduling with Resolution Adjustment algorithm (FRF-RA for short). Related parameters are set as listed in Table 2, if not stated otherwise.

We compare our method with the following classical scheduling methods:

1) Round Robin scheduling (RR): it traverses tasks in a cyclic way;
2) First Come First Serve scheduling (FCFS): it prefers to first run the earliest command.
3) Shortest Remaining Time First scheduling (SRTF): it schedules the most urgent task first. For a task, we evaluate its urgency using a deadline, which is the maximum tolerable frame interval after the last schedule.

We also simulate with the shortest job scheduling, which prefers to first run the task whose command requires the least execution time (estimated processing time plus

Authorized licensed use limited to: SHENZHEN UNIVERSITY. Downloaded on August 09,2022 at 09:18:04 UTC from IEEE Xplore. Restrictions apply.

TABLE 2
Parameter Setting for the Algorithms

Context	Parameter	Value
Formulation	$\langle \theta_r, \theta_h, \theta_d \rangle$	$\langle 63/128, 63/128, 1/64 \rangle$
Formulation	Penalty weight ϕ	0.25
RA algo	Resolution maintaining time	the time to render 1024 frames
RA algo	ξ: a threshold	0.1
FRF algo	β: the weight of $\bar{f}_i$	0.5

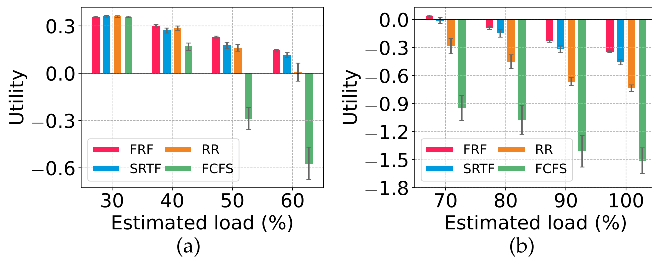


Fig. 8. The minimum utility with penalty included under various computing loads. (a) low load; (b) high load.

transmission time). But we find it performs poorly and thus omit it in the results.

In the following evaluation, each value is averaged over 50 different simulation cases. Each case is with a task assignment, a segment of trace data on processing time, and a segment of trace data on bandwidth, all of which are independently and randomly chosen. For utility and penalty, we illustrate their mean values and use error bars to represent the 95% confidence interval.

8.1 Frame Rate Fair Scheduling Algorithm

We first evaluate the frame rate fair scheduling without the resolution adjustment algorithm.

8.1.1 Improvement of Utility

As introduced in Section 5, the objective value of our problem is the minimum utility with penalty included, that is $\min_{i=1\ldots n}\tilde{u}_i$. We show this utility under various computing loads in Fig. 8. It is observed that FRF achieves the highest utility under almost all levels of load. The performance of SRTF is slightly lower than that of FRF. Both RR and FCFS perform poorly, especially under a high load. In Fig. 9, we show the minimum utility without penalty included, that is $\min_{i=1\ldots n}u_i$. It is seen that FRF achieves the highest value under all levels of load. In addition, in Fig. 10, we show the penalty of frame interval's variation, that is $\max_{i=1\ldots n}v_i$. Our method performs almost similarly to SRTF, and both of them perform better than FCFS but worse than RR.

8.1.2 Improvement of QoS Satisfaction

As introduced in Section 5, for a problem instance, if all the constraints (4b)-(4d) in the formulation (4) are satisfied, we say QoS satisfaction; otherwise, we say QoS violation. In this section, we evaluate the percentage of QoS-satisfaction (QoS-SAT for short) cases among 50 simulation cases.

It is noted that every method can meet the resolution constraint (4b), but may violate others. We show the percentage

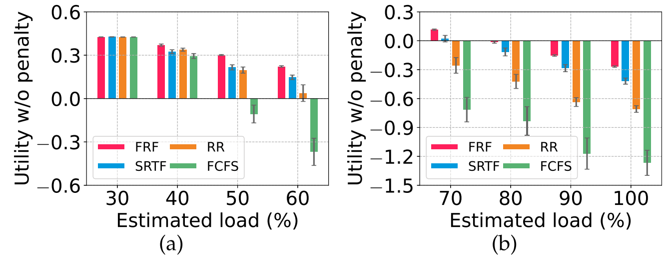


Fig. 9. The minimum utility without penalty included under various computing loads. (a) low load; (b) high load.

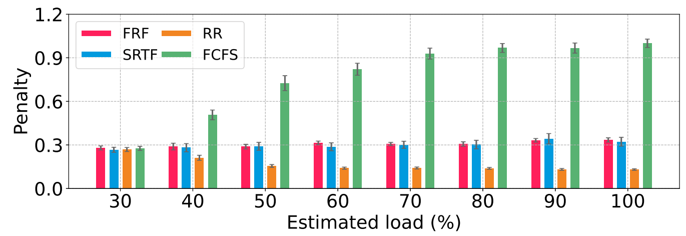


Fig. 10. The penalty of frame interval's variation under various computing loads.

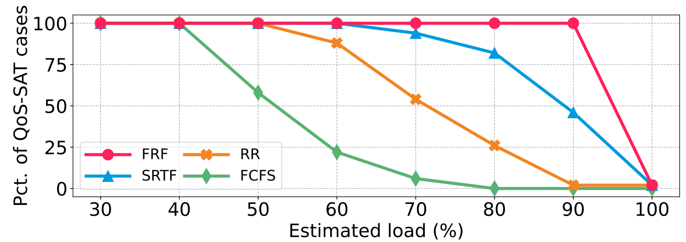


Fig. 11. The percentage of QoS-SAT cases under various computing loads.

of QoS-SAT cases in Fig. 11. Our method achieves the highest value; in contrast, all other methods experience severe violation. FRF scheduling satisfies the QoS requirements in all cases even under a high load of 90%. Furthermore, from the simulation results, we find that all the methods can meet the latency constraint (4d) when setting the latency limit as in (37). In this situation, all the QoS-violation cases violate the frame interval constraint (4c).

8.2 Resolution Adjustment Algorithm

Our resolution adjustment algorithm can be combined with any scheduling algorithm. In this section, we evaluate the impact of the RA algorithm on all methods.

8.2.1 Improvement of Utility

We join the RA algorithm with each of the methods and evaluate the impact on the utility. As shown in Fig. 12, our method FRF-RA performs the best under all loads. We illustrate the utility gain compared to the original method in Fig. 13. It is observed that under a load below 50% the utility considerately increases for almost every method. Roughly speaking, the lower the load, the higher the gain because of more free computing power.

8.2.2 Impact on QoS Satisfaction

Next, we evaluate the impact of the RA algorithm on QoS satisfaction. We illustrate the percentage of QoS-SAT cases for

Authorized licensed use limited to: SHENZHEN UNIVERSITY. Downloaded on August 09,2022 at 09:18:04 UTC from IEEE Xplore. Restrictions apply.

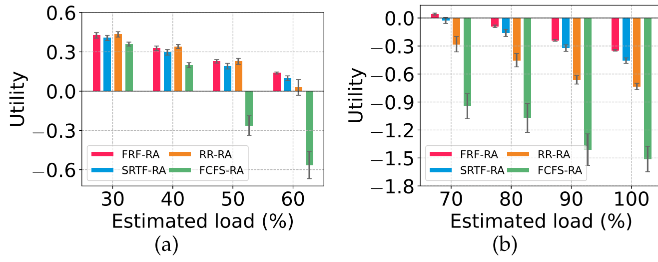


Fig. 12. The minimum utility with penalty included under various computing loads when joining the RA algorithm with each of the methods. (a) low load; (b) high load.

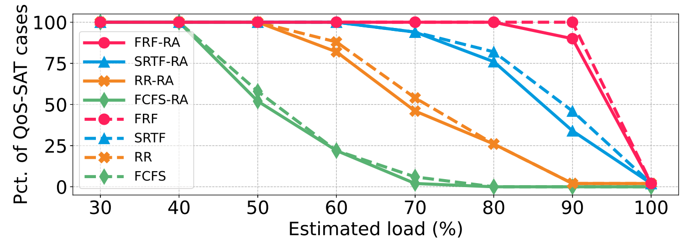


Fig. 14. The percentage of QoS-SAT cases with and without the resolution adjustment algorithm.

each method with and without the RA algorithm in Fig. 14. It is observed that the RA algorithm degrades the performance of QoS satisfaction slightly under the high loads. This degradation is the cost of the improvement in utility under the low loads as shown in Fig. 13. It is mainly due to the trial on resolution arrangements in the RA algorithm.

8.3 Ablation Study

In this section, we evaluate the effectiveness of different design choices of our proposed algorithm.

8.3.1 Effectiveness of Optimizing Scheduling Sequence

Recall that FRF consists of two critical issues: the weighted max-min frame rate problem and the scheduling sequence problem. In this section, we demonstrate that optimizing the scheduling sequence is effective. We compare our methods (FRF and FRF-RA) with the versions without optimizing the scheduling sequence (FRF-wo-Order and FRF-RA-wo-Order). In the comparison methods, instead of calling Algorithm 4, we generate a scheduling sequence naively in a round-robin way.

Fig. 15a illustrates the percentage of QoS-SAT cases, our methods improve the percentage by up to 50 points over their counterparts under the load of 90%. Fig. 15b illustrates the penalty of frame interval's variation for these methods. It is obvious that both our methods achieve lower penalties than their counterparts. This implies that our optimization of the scheduling sequence is effective in smoothing frame intervals. Fig. 15c illustrates the utility with penalty included, and our methods achieve higher utility than their counterparts.

8.3.2 Effectiveness of Utility Function u(x)

As mentioned in Section 5.2, the traditional utility function $\log(x)$ cannot introduce sufficient penalty for performance violation to guarantee QoS. Here, we compare $\log(x)$ and the function $u(x)$ defined in (6). In order to demonstrate their difference, we set the latency limit lower as below:

$$d_{\max} = \min\{80, \text{round}\left(\frac{t_p(r_{\min}) + t_x(r_{\min})}{10}\right) \cdot 10 + 20\}. \quad (40)$$

Fig. 16a shows the performance of various methods with either $u(x)$ or $\log(x)$. It is observed that $u(x)$ can improve QoS satisfaction. In particular, when FRF-RA is used under the load of 30%, taking $u(x)$ satisfies the QoS requirements in all cases; in contrast, taking $\log(x)$ only satisfies the QoS requirements in 86% cases.

In addition, it is noticed that in the resolution adjustment algorithm a convergence condition plays an important role in satisfying the QoS requirements, that is the evaluation of QoS constraints (line 16 in Algorithm 1). Thus, we remove this convergence condition and compare the performance of $u(x)$ and $\log(x)$ again. In this situation, we set the latency limit as in (37). Fig. 16b illustrates the results. It is observed that $\log(x)$ is extremely worse than $u(x)$ in meeting the QoS requirements. These simulation results are consistent with our motivation for designing the utility function.

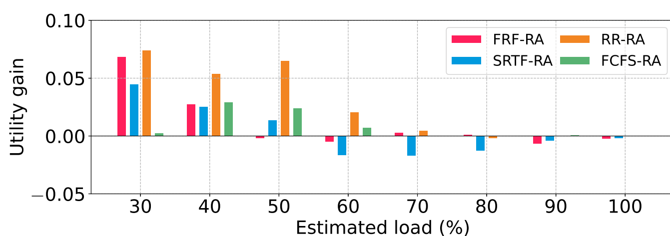


Fig. 13. The utility gain brought by joining the RA algorithm with each of the methods.

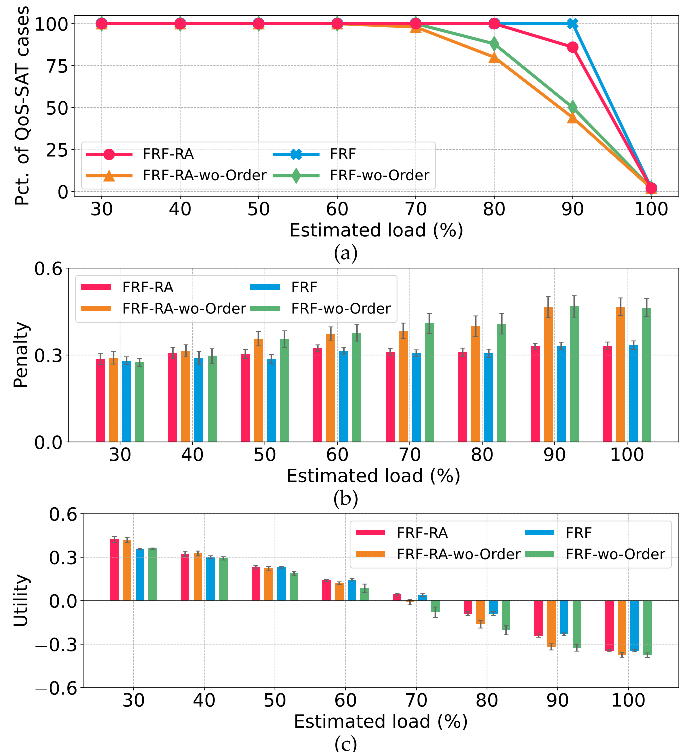


Fig. 15. The comparison of our methods with the versions without optimizing scheduling sequence. (a) the percentage of QoS-SAT cases; (b) penalty; (c) utility.

Authorized licensed use limited to: SHENZHEN UNIVERSITY. Downloaded on August 09,2022 at 09:18:04 UTC from IEEE Xplore. Restrictions apply.

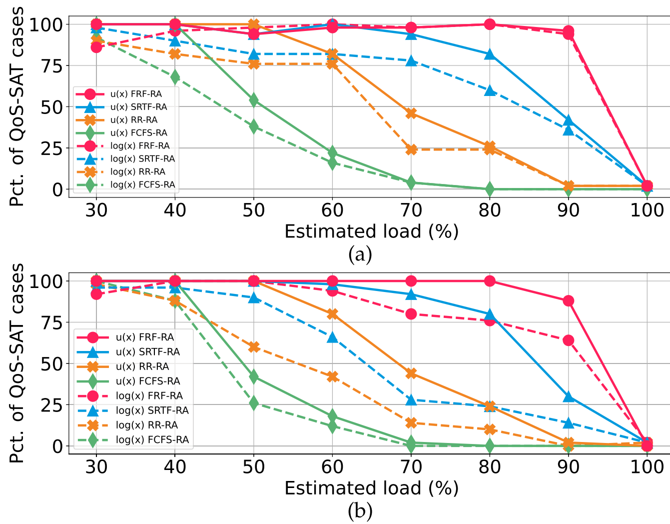


Fig. 16. The performance comparison of the utility function $u(x)$ defined in (6) and $\log(x)$. (a) setting the latency limit as in (40); (b) setting the latency limit as in (37), and removing the convergence condition of the evaluation on QoS constraints from the RA (line 16 in Algorithm 1).

8.3.3 *Effectiveness of Accumulating Q_i in FRF*

As mentioned in Section 6.2.1, in Algorithm 2, we get m_i by accumulating Q_i in D_i and rounding down D_i, rather than directly rounding down Q_i. Here, we compare these two operations. The results are illustrated in Fig. 17. The comparison methods (FRF-wo-Acc and FRF-RA-wo-Acc) perform worse than their counterparts under the high loads above 70%, both in terms of QoS-SAT and utility.

8.4 Evaluation of θ_d

In this section, we evaluate the impact of the weight θ_d on performance. We vary θ_d between three values: 0, 1/64 and 1/8, and let $\theta_r = \theta_h = (1-\theta_d)/2$. As shown in Fig. 18a, as θ_d increases from 0 to 1/8, the utility gain brought by the RA algorithm decreases. The reason is that promoting resolution increases latency and is restrained by raising the weight on latency. We also illustrate the percentage of QoS-SAT cases in Fig. 18b. It is observed that as θ_d increases, the percentage value increases slightly.

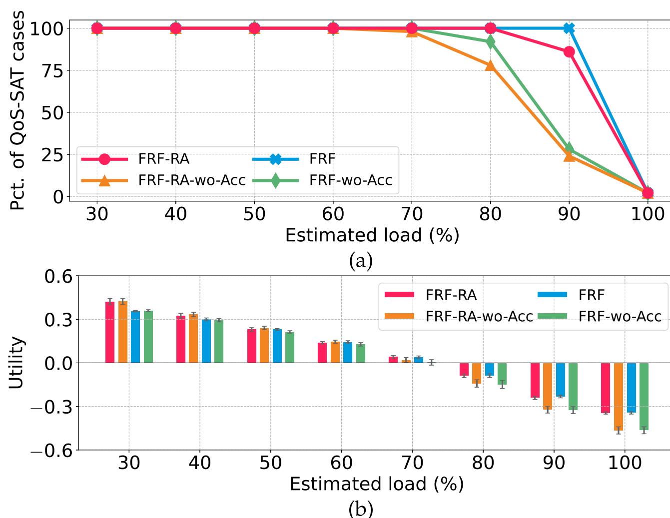


Fig. 17. The comparison of our methods with the versions without accumulating Q_i. (a) the percentage of QoS-SAT cases; (b) utility.

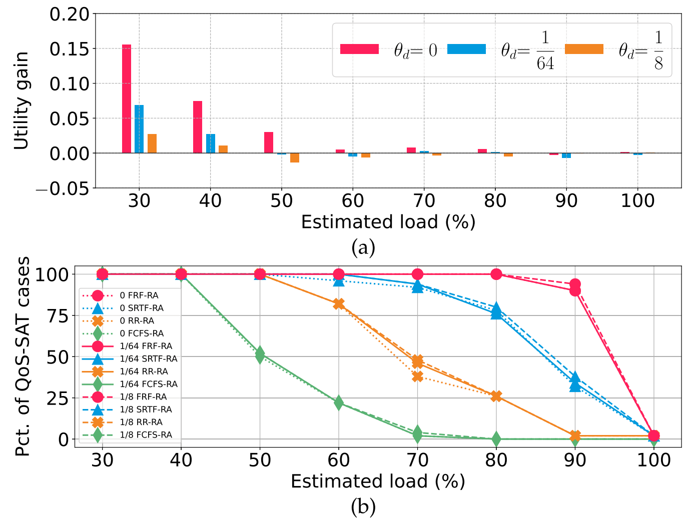


Fig. 18. The performance when varying θ_d. (a) the utility gain of the RA algorithm; (b) the percentage of QoS-SAT cases.

8.5 Evaluation of θ_r

In this section, we evaluate the impact of the weight θ_r on performance. We vary θ_r between three different values: $2/3(1-\theta_d)$, $1/2(1-\theta_d)$ and $1/3(1-\theta_d)$, and let $\theta_d = 1/64$ and $\theta_h = 1-\theta_d-\theta_r$. As shown in Fig. 19a, as θ_r increases, the utility gain brought by the RA algorithm increases. The reason is that when the resolution has a higher weight on the total utility, the benefit brought by elevating resolution is larger than the loss caused by increased latency. We also illustrate the percentage of QoS-SAT cases in Fig. 19b. It is observed that the percentage values in all three situations are almost close.

8.6 Evaluation of ξ

Recall that in the RA algorithm one of the convergence conditions is that the drop of the utility relative to the best value

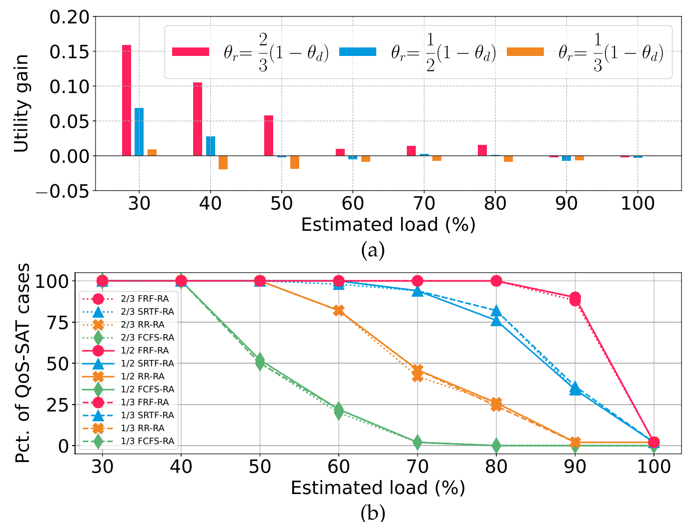


Fig. 19. The performance when varying θ_r. (a) the utility gain of the RA algorithm; (b) the percentage of QoS-SAT cases.

Authorized licensed use limited to: SHENZHEN UNIVERSITY. Downloaded on August 09,2022 at 09:18:04 UTC from IEEE Xplore. Restrictions apply.

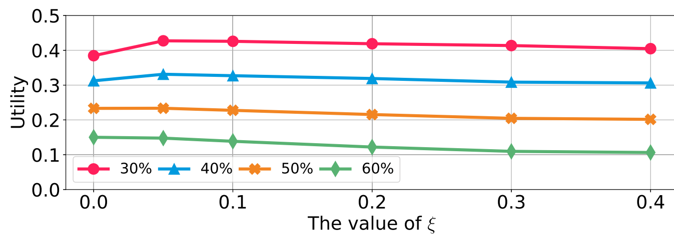


Fig. 20. The utility versus the threshold ξ. Each line corresponds to a level of load. Some.

found in history exceeds a threshold (i.e., ξ). We evaluate the impact of the threshold on performance by varying its value. In Fig. 20, we illustrate the results under four levels of load from 30% to 60%, since under a higher load there is no extra computing power to improve resolution. It is observed that when ξ is zero, the utility under a load of 30% is very low, although there is abundant computing power in that situation. As ξ increases, utility increases. But further increasing ξ may decrease utility due to trying out bad arrangements. Thus, in our simulation, we set ξ to 0.1.

9 Conclusion

In this paper, we investigate the QoS-aware rendering task scheduling problem in edge computing, that is to make real-time decisions on which task to execute in which resolution such that user requirements are met and user performance is maximized simultaneously. We formulate the problem into a QoS constrained max-min utility problem. We propose an efficient scheduling algorithm to solve the problem, which consists of a resolution adjustment algorithm and a frame rate fair scheduling algorithm. Our simulations based on actual rendering data demonstrate that our method can improve utility and QoS satisfaction compared with competing QoS-oblivious methods.

References

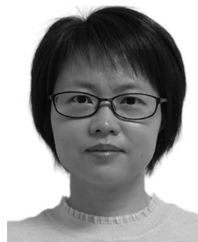
[1] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 3, pp. 1628–1656, Jul.–Sep. 2017.
[2] L. Lin, X. Liao, H. Jin, and P. Li, "Computation offloading toward edge computing," *Proc. IEEE*, vol. 107, no. 8, pp. 1584–1607, Aug. 2019.
[3] R. Shea, J. Liu, E. C.-H. Ngai, and Y. Cui, "Cloud gaming: Architecture and performance," *IEEE Netw.*, vol. 27, no. 4, pp. 16–21, Jul./Aug. 2013.
[4] OnLive, 2022. [Online]. Available: https://en.wikipedia.org/wiki/OnLive
[5] GeForce Now, 2022. [Online]. Available: https://www.nvidia.com/en-us/geforce-now/
[6] CloudXR, 2022. [Online]. Available: https://www.nvidia.com/en-us/design-visualization/solutions/cloud-xr/
[7] X. Zhang *et al.*, "Improving cloud gaming experience through mobile edge computing," *IEEE Wireless Commun.*, vol. 26, no. 4, pp. 178–183, Aug. 2019.
[8] S. Sukhmani, M. Sadeghi, M. Erol-Kantarci, and A. El Saddik, "Edge caching and computing in 5G for mobile AR/VR and tactile internet," *IEEE MultiMedia*, vol. 26, no. 1, pp. 21–30, Jan.–Mar. 2019.
[9] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 455–466.
[10] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 323–336.
[11] S. Wang, X. Li, Q. Z. Sheng, R. Ruiz, J. Zhang, and A. Beheshti, "Multi-queue request scheduling for profit maximization in IaaS clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 11, pp. 2838–2851, Nov. 2021.
[12] J. Zhu, X. Li, R. Ruiz, W. Li, H. Huang, and A. Y. Zomaya, "Scheduling periodical multi-stage jobs with fuzziness to elastic cloud resources," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 12, pp. 2819–2833, Dec. 2020.
[13] J. Zhu, X. Li, R. Ruiz, and X. Xu, "Scheduling stochastic multi-stage jobs to elastic hybrid cloud resources," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 6, pp. 1401–1415, Jun. 2018.
[14] J. Liu, J. Ren, Y. Zhang, X. Peng, Y. Zhang, and Y. Yang, "Efficient dependent task offloading for multiple applications in MEC-cloud system," *IEEE Trans. Mobile Comput.*, to be published, doi: 10.1109/TMC.2021.3119200.
[15] S. Yue *et al.*, "TODG: Distributed task offloading with delay guarantees for edge computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 7, pp. 1650–1665, Jul. 2022.
[16] W. Liu, J. Cao, L. Yang, L. Xu, X. Qiu, and J. Li, "AppBooster: Boosting the performance of interactive mobile applications with computation offloading and parameter tuning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 6, pp. 1593–1606, Jun. 2017.
[17] C. Reaño, F. Silla, D. S. Nikolopoulos, and B. Varghese, "Intra-node memory safe GPU co-scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 5, pp. 1089–1102, May 2018.
[18] Y. Zhang, P. Qu, J. Cihang, and W. Zheng, "A cloud gaming system based on user-level virtualization and its resource scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 5, pp. 1239–1252, May 2016.
[19] Z. Qi, J. Yao, C. Zhang, M. Yu, Z. Yang, and H. Guan, "VGRIS: Virtualized GPU resource isolation and scheduling in cloud gaming," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 2, Jul. 2014, Art. no. 17.
[20] C. Zhang, J. Yao, Z. Qi, M. Yu, and H. Guan, "vGASA: Adaptive scheduling algorithm of virtualized GPU resource in cloud gaming," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 11, pp. 3036–3045, Nov. 2014.
[21] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proc. ACM Special Int. Group Data Commun.*, 2019, pp. 270–288.
[22] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni, "GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters," in *Proc. 12th USENIX Conf. Oper. Syst. Des. Implementation*, 2016, pp. 81–97.
[23] Stadia, 2022. [Online]. Available: https://stadia.google.com/
[24] M. Viitanen, J. Vanne, T. D. Hämäläinen, and A. Kulmala, "Low latency edge rendering scheme for interactive 360 degree virtual reality gaming," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 1557–1560.
[25] L. Liu, H. Li, and M. Gruteser, "Edge assisted real-time object detection for mobile augmented reality," in *Proc. 25th Annu. Int. Conf. Mobile Comput. Netw.*, 2019, Art. no. 25.
[26] L. Liu *et al.*, "Cutting the cord: Designing a high-quality untethered VR system with low latency remote rendering," in *Proc. 16th Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2018, pp. 68–80.
[27] X. Liao *et al.*, "LiveRender: A cloud gaming system based on compressed graphics streaming," *IEEE/ACM Trans. Netw.*, vol. 24, no. 4, pp. 2128–2139, Aug. 2016.
[28] Augmented and virtual reality: The first wave of 5G killer apps, 2017. [Online]. Available: https://www.qualcomm.com/media/documents/files/augmented-and-virtual-reality-the-first-wave-of-5g-killer-apps.pdf
[29] E. Bastug, M. Bennis, M. Medard, and M. Debbah, "Toward interconnected virtual reality: Opportunities, challenges, and enablers," *IEEE Commun. Mag.*, vol. 55, no. 6, pp. 110–117, Jun. 2017.
[30] N. C. Luong, P. Wang, D. Niyato, Y. Wen, and Z. Han, "Resource management in cloud networking using economic analysis and pricing models: A survey," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 2, pp. 954–1001, Apr.–Jun. 2017.
[31] Y. Zhang, X. Lan, J. Ren, and L. Cai, "Efficient computing resource sharing for mobile edge-cloud computing networks," *IEEE/ACM Trans. Netw.*, vol. 28, no. 3, pp. 1227–1240, Jun. 2020.
[32] R. Xie, J. Fang, J. Yao, X. Jia, and K. Wu, "Sharing-aware task offloading of remote rendering for interactive applications in mobile edge computing," *IEEE Trans. Cloud Comput.*, to be published, doi: 10.1109/TCC.2021.3127345.

Authorized licensed use limited to: SHENZHEN UNIVERSITY. Downloaded on August 09,2022 at 09:18:04 UTC from IEEE Xplore. Restrictions apply.

[33] Y. T. Hou, H. H.-Y. Tzeng, and S. S. Panwar, "A generalized max-min rate allocation policy and its distributed implementation using the ABR flow control mechanism," in *Proc. IEEE Conf. Comput. Commun.*, 1998, pp. 1366–1375.
[34] B. Radunovic and J.-Y. Le Boudec, "A unified framework for max-min and min-max fairness with applications," *IEEE/ACM Trans. Netw.*, vol. 15, no. 5, pp. 1073–1083, Oct. 2007.
[35] Unity, 2022. [Online]. Available: https://unity.com/
[36] ArchVizPRO interior, vol. 7, 2020. [Online]. Available: https://assetstore.unity.com/packages/3d/environments/urban/archvizpro-interior-vol-7-155448
[37] Continuous speed test tool, 2022. [Online]. Available: http://startrinity.com/InternetQuality/ContinuousBandwidthTester.aspx

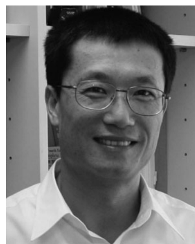
Ruitao Xie received the BEng degree from the Beijing University of Posts and Telecommunications, Beijing, China, in 2008, and the PhD degree in computer science from the City University of Hong Kong, Hong Kong, in 2014. She is currently an assistant professor with the College of Computer Science and Software Engineering, Shenzhen University. Her research interests include edge computing, AI networking, cloud computing, and distributed systems.

Junhong Fang received the BEng degree in computer science from the China University of Mining and Technology, Xuzhou, China, in 2020. He is currently working toward the graduate degree in the College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China.



Junmei Yao received the BEng and MEng degrees from the Harbin Institute of Technology, Harbin, China, in 2003 and 2005, respectively, and the PhD degree in computer science from the Hong Kong Polytechnic University, Hong Kong, in 2016. She is currently an assistant professor with the College of Computer Science and Software Engineering, Shenzhen University, China. Her research interests include wireless networks, wireless communications, and mobile computing.

Kai Liu (Senior Member, IEEE) received the PhD degree in computer science from the City University of Hong Kong, Hong Kong, in 2011. From December 2010 to May 2011, he was a visiting scholar with the Department of Computer Science, University of Virginia, USA. From 2011 to 2014, he was a postdoctoral fellow with Singapore Nanyang Technological University, City University of Hong Kong, and Hong Kong Baptist University. He is currently a professor with the College of Computer Science, Chongqing University, China. His research interests include Internet of vehicles, mobile edge computing, pervasive computing, and intelligent transportation systems.



Xiaohua Jia (Fellow, IEEE) received the BSc and MEng degrees from the University of Science and Technology of China, Hefei, China, in 1984 and 1987, respectively, and the DSc degree in information science from the University of Tokyo, Tokyo, Japan, in 1991. He is currently chair professor with the Department of Computer Science, City University of Hong Kong. His research interests include cloud computing and distributed systems, data security and privacy, computer networks, and mobile computing. He is an editor of the *IEEE Transactions on Computers* (2021 – present), *IEEE Internet of Things* (2013-2018), *IEEE Transactions on Parallel and Distributed Systems* (2006-2009), *Journal of World Wide Web*, *Journal of Combinatorial Optimization*, etc. He is the general chair of ACM MobiHoc 2008, TPC co-chair of IEEE GlobeCom 2010 – Ad Hoc and Sensor Networking Symp, area-chair of IEEE INFOCOM 2015-2017, chair of ACM ICN 2019, and chair of IEEE ICDCS 2023. He is a fellow of the IEEE (Computer Society) and distinguished member of the ACM.

Kaishun Wu received the PhD degree in computer science and engineering from HKUST, Hong Kong, in 2011. After that, he worked as a research assistant professor with HKUST. In 2013, he joined SZU as a distinguished professor. He has co-authored two books and published more than 100 high quality research papers in international leading journals and primer conferences, like the *IEEE Transactions on Mobile Computing*, *IEEE Transactions on Parallel and Distributed Systems*, ACM MobiCom, IEEE INFOCOM. He is the inventor of six US and more than 90 Chinese pending patent. He received 2012 Hong Kong Young Scientist Award, 2014 Hong Kong ICT awards: Best Innovation and 2014 IEEE ComSoc Asia-Pacific Outstanding Young Researcher Award. He is an IET fellow.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: SHENZHEN UNIVERSITY. Downloaded on August 09,2022 at 09:18:04 UTC from IEEE Xplore. Restrictions apply.